TEXAS INSTRUMENTS

SCOPE™ Testability Products

Applications Guide

1990

Semiconductor Group

SCOPE™ Testability Products Applications Guide

Design Automation — Semiconductor Group
Texas Instruments

TEXAS INSTRUMENTS

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to or to discontinue any semiconductor product or service identified in this publication without notice. TI advises its customers to obtain the latest version of the relevant information to verify, before placing orders, that the information being relied upon is current.

TI warrants performance of its semiconductor products to current specifications in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Unless mandated by government requirements, specific testing of all parameters of each device is not necessarily performed.

TI assumes no liability for TI applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

Texas Instruments products are not intended for use in life-support appliances, devices, or systems. Use of a TI product in such applications without the written consent of the appropriate TI officer is prohibited.

WARNING

This equipment is intended for use in a laboratory test environment only. It generates, uses, and can radiate radio frequency energy and has not been tested for compliance with the limits of computing devices pursuant to subpart J of part 15 of FCC rules, which are designed to provide reasonable protection against radio frequency interference. Operation of this equipment in other environments may cause interference with radio communications, in which case the user at his own expense will be required to take whatever measures may be required to correct this interference.

TRADEMARKS

ASSET is a trademark of Texas Instruments Incorporated.

IBM is a trademark of International Business Machines Corporation.

MegaModule is a trademark of Texas Instruments Incorporated.

PC-AT is a trademark of International Business Machines Corporation.

SCOPE is a trademark of Texas Instruments Incorporated.

TRI-STATE® is a registered trademark of National Semiconductor Corporation.

Contents

List of Illustrations

Design Tradeoffs When Implementing IEEE 1149.1, *by Wayne Daniel*

Hardware and Software Integration and Debugging Using ASSET, *by Daniel R. Fusting*

System Testability Using Standard Logic, *by R. J. Morgan*

Using the ASSET Constraint Checker, *by Jeffrey Aubert*

"What's an LFSR?," *by John Koeter*

List of Tables

SCOPE Applications Guide Overview

by Javier Romeu

The rapid growth in the size of VLSI (Very Large Scale Integration) devices, the complexity of printed circuit boards, tighter design geometries, and advances in surface mount technologies are making conventional test methods obsolete.

Traditional board-level and device-level testing consumes a great deal of time and requires special algorithms and complex Automated Test Equipment (ATE) for each type of board or device. This results in increasing costs and development time. Many existing systems check only some of the components or circuits, and most testing is done with the device removed from the target board. The extensive testing necessitated by the heightening reliability standards and performance standards in the defense, aerospace, automotive, computer, and communications industries can delay the market introduction of products, disrupt JIT (Just In Time) manufacturing flows, and limit the productivity of standard ATE operations. An innovative approach to this problem is to incorporate design-for-test techniques that allow embedded testing to be performed.

Testing can then be done while the component or circuit is undergoing standard burn-in or quality control procedures and any number of times during its normal operation.

SCOPE (System Controllability, Observability, and Partitioning Environment), TI's family of Design-For-Test products based on the IEEE 1149.1 standard (also known as JTAG), provides a fresh look at testing procedures. TI's boundary scan SCOPE product line supports the engineer who designs ASICs (Application-Specific Integrated Circuits) or circuit boards. Additionally, SCOPE's boundary-scan test features can be configured for sophisticated built-in-test and debug operations using programs such as TI's ASSET (Advanced Support System for Emulation and Test) software. Many ICs or boards may be tested together using the serial IEEE 1149.1 test bus under the control of ASSET. TI's SCOPE hardware and software products make use of the IEEE 1149.1 boundary scan and test access port specifications and support circuit status analysis, data sampling, pseudo-random pattern generation, parallel signature analysis, and built-in-test.

As is the case with new design concepts and standards, designers face the challenge of learning how to implement these specifications and reap the benefits of their advantages. The application notes in this book are intended to give the designer a working familiarity with TI's Design-For-Test SCOPE products. The papers are the work of engineers and designers from many different areas of TI, thus, some concepts are mentioned from different design viewpoints.

Each article covers at least one important aspect of Design-For-Testing. Topics such as partitioning, boundary scan, TI's SCOPE octals, TI's ASSET, opcodes, SCOPE ASIC cells, test and debugging, and constraint checking are covered. Articles are listed alphabetically by title, and the index points to articles containing significant information about certain topics.

These application articles cover the current state of boundary scan testing and TI's SCOPE products. Future updated editions are planned that will cover the evolving area of design-for-test and the growing SCOPE product line. We welcome your feedback and suggestions for future editions of this applications guide. If you require additional information about all TI testability products, contact the nearest TI Field Sales Office or call the customer support numbers listed in the back of this guide.

ASSET Configuration Files

by Bill Weiss

Introduction

A ring or scan path configuration consists of devices connected to their neighbors by the serial scan path. Data or instructions directed to one or more devices must go through all other devices on the scan path. This requires that the managing software knows at all times what devices are currently in the scan path and the current scan path length for each device. This information is conveyed to ASSET via configuration files. In this paper, the use, contents, and format of ASSET device and board/module configuration files are illustrated. Two examples are used to explain the creation of configuration files and their translation into ASSET code. A floating point translator ASIC is developed for the Device Configuration file. The General Demonstration Module is used for the Module configuration file. Two software programs that aid in the development and translation of configuration files are also discussed. The first program is the Configuration Editor, which allows rapid development of device and module configuration files by removing the burden of remembering the file syntax from the user. Once the configuration file has been created, the Configuration Translator can be used to generate ASSET code automatically.

ASSET Configuration Files

ASSET requires a configuration file to be defined for all scannable functions in the target system. ASSET therefore supports two types of configuration files: a device configuration file to describe IEEE 1149.1 devices, and a board or module configuration file to describe boards or higher level modules that contain 1149.1 devices or modules. Each of the configuration files has a default filename extension. The device configuration filename extension is .dcf, and the module configuration filename extension is .mcf. The source format for a configuration file is ASCII.

The Configuration Editor or a stand-alone text editor can be used to create the configuration file. However, the Configuration Editor is recommended when first learning how to create configuration files. Each type of ASSET configuration file requires a minimum amount of information. A device configuration file requires a class declaration describing the name of the device and the type of scan architecture the device supports, FIELD declarations with an instruction register, bypass register and boundary registers, a path declaration with paths for the instruction register, bypass register and the ordering of the boundary registers, and a command declaration with EXTEST, SAMPLE, BYPASS (1149.1 commands) defined. The module configuration file requires a class declaration describing the name and type of configuration file, component declarations listing the type and name of the scannable devices on the board, and a path declaration describing the ordering of the components on the module. A more detailed description of ASSET configuration files is given in the sections that follow.

Some general syntactical rules apply to all configuration files. Names in configuration files must be unique and from 1–16 characters in length. They may contain alphanumeric characters or an underscore with the first character being alphabetic. Configuration files are not case sensitive but will be converted to lowercase when processed by the Configuration Translator. All statements in a configuration file are terminated with a semicolon. Comments may be added to a configuration file using a stand-alone text editor. The syntax for comments may be in one of two forms:

```
//This is a comment to end of line
            OR
/*This is also a comment */.
```

Device Configuration Files

Before accessing and analyzing the registers in a device, a device configuration file for that device must be developed. In

this section we will develop the configuration file for an IEEE Floating Point Translator ASIC (Application-Specific Integrated Circuit), part number CF93279.

Class Declaration

Class declaration defines the name of the device and the type of scan architecture the device supports. The syntax for a class declaration is:

CLASS device_name : scan_architecture;

ASSET currently only supports the 1149.1 scan architecture. ASSET requires that the name of the device matches the name of the configuration file. In the example configuration file, the class declaration would look like:

CLASS CF93279:device;

Field Declarations

A field declaration defines the scannable registers within a device. The syntax for a field declaration is:

FIELD fieldname1(msb,lsb),....,fieldnameN(msb,lsb);

The key word FIELD begins the declaration and may include one or more field names. The msb variable stands for the most significant bit in the register while the lsb variable stands for the least significant bit in the register. The maximum field size is 10000 hex. ASSET device configuration files reserve the names "inst" and "byp" to represent the instruction and bypass registers respectively. The IEEE 1149.1 standard specifies the instruction register have a minimum length of 2 bits. In the example device, the instruction register is 8 bits in length. The bypass register is always 1-bit wide. The Floating Point Translator has the following additional scannable registers: DA data bus, DB data bus, PIPE, DIR, OEZ, WAIT, NAN, CLK. The two data buses are 32-bits wide and the remaining registers are 1-bit wide. The field declarations for the Floating Point Translator are:

```
FIELD inst(7,0);
FIELD byp(0,0);
FIELD clk(0,0);
FIELD pipe(0,0);
FIELD oez(0,0);
FIELD dir(0,0);
FIELD wait(0,0);
FIELD nan(0,0);
FIELD da(31,0);
FIELD db(31,0);
```

Path Declarations

The path declaration represents the physical layout of the scan cells in the scan device. A path declaration consists of a DEVPATH key word and a pathname followed by a list of the field items in the path. The path declaration may be a multiple line entry. The syntax for a path declaration is:

DEVPATH pathname field_item1[, field_item2, . . . , field_itemN]; where field_item has the format: field_name (bit_number, bit_number).

The ordering of the scan cells within the device is crucial. Scan cells should be ordered such that the cell closest to TDO (Test Data Out) is defined first. In the example, the path declarations for the instruction and bypass registers are simple. The names "ipath" and "bypath" are reserved for the instruction register and bypass register path names. They each have one field in the path as shown:

```
DEVPATH ipath inst(7,0);
DEVPATH bypath byp(0,0);
```

Figure 1 shows the scan path for the boundary registers in our example. The path declaration for the boundary scan path would look like:

```
DEVPATH boundary
  db(0,31), da(0,31), wait(0,0), nan(0,0),
  oez(0,0), clk(0,0), pipe(0,0), dir(0,0);
```

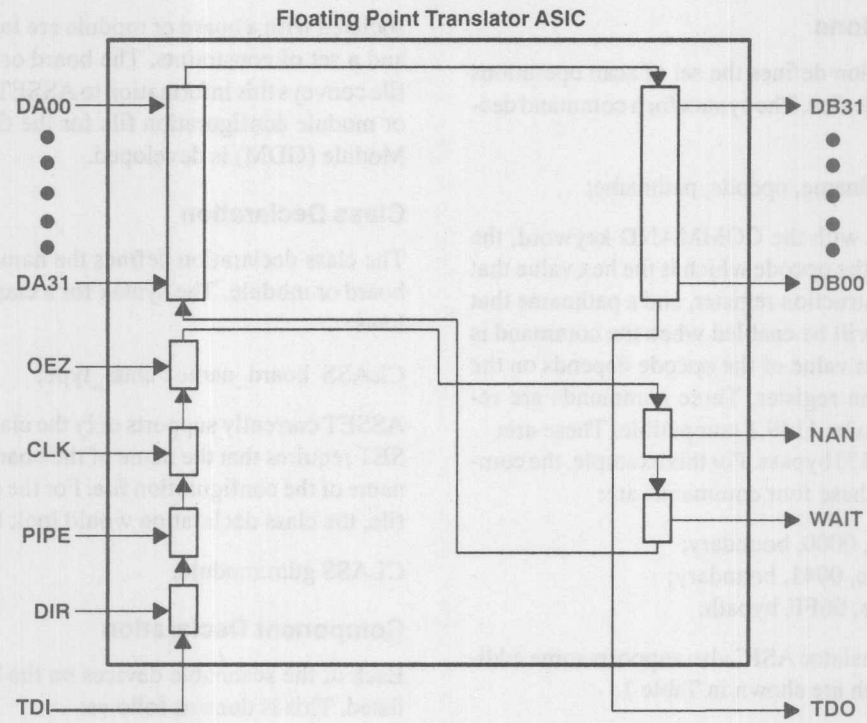


Figure 1. Boundary Scan Path Diagram

Notice in the field declarations the registers were declared with the lsb being zero and the msb being the nonzero value while in the path declarations, some of the registers are defined just the opposite. This is because the path declarations must reflect the physical layout of the registers in the device. A field may be split into separate fields in a path declaration. Consider two fields declarations; FIELD addr(15,0) and dataio(3,0). If the addr field physically splits the dataio field into two fields of 2 bits each on the scan path, the path declaration can be written as follows:

DEVPATH pathname dataio(3,2), addr(15,0), dataio(1,0);. See Figure 2.

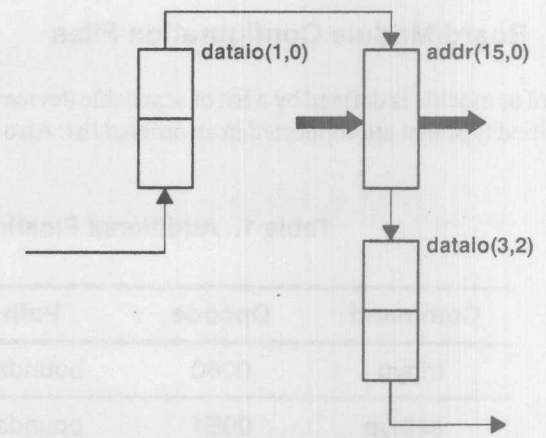


Figure 2. Example of a Split Field in a Path

Command Declarations

The command declaration defines the set of scan operations that are supported by a device. The syntax for a command declaration is:

COMMAND commandname, opcode, pathname;

The declaration begins with the COMMAND keyword, the name of the command, the opcode which is the hex value that will be placed in the instruction register, and a pathname that specifies the path that will be enabled when the command is initiated. The maximum value of the opcode depends on the length of the instruction register. Three commands are required for the device to be 1149.1 compatible. These are: 1) extest, 2) sample, and 3) bypass. For this example, the command declarations for these four commands are:

```
COMMAND extest, 0000, boundary;
COMMAND sample, 0041, boundary;
COMMAND bypass, 00FF, bypath;
```

The Floating Point Translator ASIC also supports some additional commands, which are shown in Table 1.

The path declarations for these commands are as follows:

```
COMMAND  idcode, 0081, bypath;
COMMAND tribyp, 0060, bypath;
COMMAND setbyp, 00E1, bypath;
COMMAND readbn, 0050, boundary;
COMMAND readbt, 00D1, boundary;
COMMAND celltst, 0030, boundary;
```

Board/Module Configuration Files

A board or module is defined by a list of scannable devices of a specified type that are connected in an ordered list. Also associated with a board or module are logical groups of signals and a set of constraints. The board or module configuration file conveys this information to ASSET. In this section a board or module configuration file for the General Demonstration Module (GDM) is developed.

Class Declaration

The class declaration defines the name and class type of the board or module. The syntax for a class declaration is as follows:

CLASS board_name : class_type;

ASSET currently supports only the class type of module. ASSET requires that the name of the board or module match the name of the configuration file. For the example configuration file, the class declaration would look like:

CLASS gdm:module;

Component Declaration

Each of the scannable devices on the board/module must be listed. This is done as follows:

device_name
instance_name1[,instance_name2,...,instance_nameN];

The device_name refers to the device configuration in another file. Each instance_name must be unique. A scan path diagram for the GDM is shown in Figure 3.

It shows the scannable devices on the GDM and their device_names. For the GDM, the component declarations would look like this:

```
T8373  u22;
T8244  u1, u8, u9, u19, u21;
CF93279 u20;
```

Table 1. Additional Floating Point Translator ASIC Commands

Command	Opcode	Path	Description
tribyp	0060	boundary	Tristate outputs and bypass in test mode
setbyp	00E1	boundary	Set outputs and bypass in test mode
readbn	0050	boundary	Read boundary register in normal mode
readbt	00D1	boundary	Read boundary register in test mode
celltst	0030	boundary	Run cell self test in normal mode

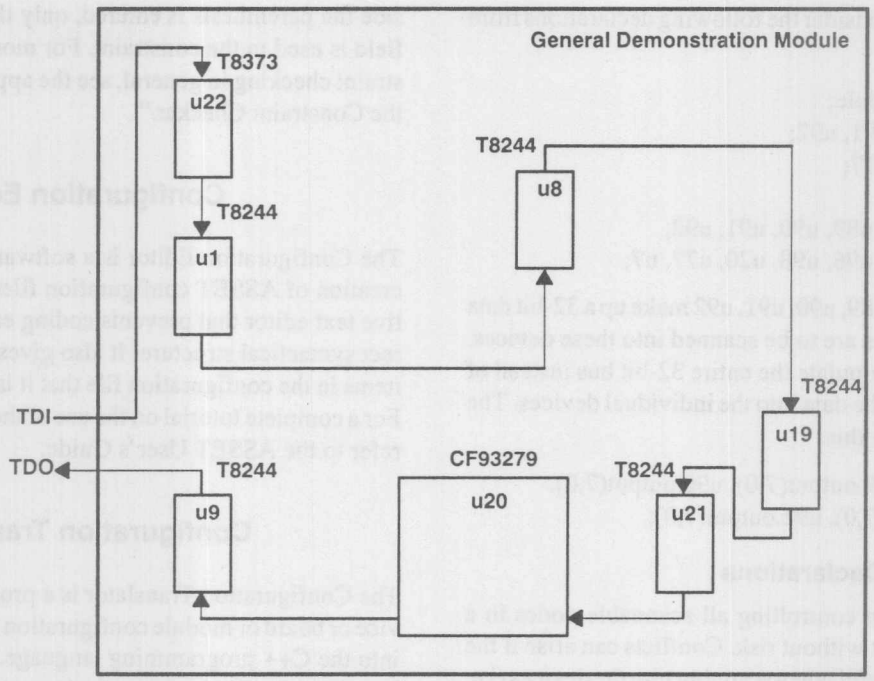


Figure 3. GDM Scan Path Diagram

Notice that the Floating Point Translator ASIC developed in the previous section is used on the board.

Path Declaration

The path declaration represents the physical layout of the scannable devices on the board or module. A path declaration consists of the MODPATH keyword and a pathname followed by a list of the devices in the path. The path declaration may be a multiple line entry, if required. The syntax for a path declaration is:

MODPATH pathname device1[, device2, ... , deviceN];

The ordering of the devices on the scan path is crucial. Order the devices such that the device closest to TDO (Test Data Out) is defined first. In the example, the path declaration for the GDM is:

MODPATH path1 u9, u20, u21, u19, u8, u1, u22;

Signal Declarations

It may be desirable to group different fields from different devices into a single convenient entity or manipulate individual bits within a field using a single logical name. A signal allows such entities to be defined, which can then be manipulated directly. A signal declaration has the following syntax:

SIGNAL
signal_name device_name.field_name1(msb,lsb)
[, device_name.field_name2(msb,lsb), ... ,
device_name.field_nameN(msb,lsb)];

In the GDM, four signals have been declared:

SIGNAL da30 u19.input(7,7);
SIGNAL da31 u19.input(6,6);
SIGNAL da30d u19.input(5,5);
SIGNAL hexdis u9.output (0,3);

The first three signal declarations define individual bits to be manipulated, while the fourth signal declaration simply reorders the four lsb's of the field.

As another example, consider the following declarations from a memory board:

```
CLASS mem : module;
T8244  u89, u90, u91, u92;
T8245  u96, u98, u77;
T8374  u20, u7;
MODPATH path1  u89, u90, u91, u92;
MODPATH path2  u96, u98, u20, u77, u7;
```

Suppose that devices u89, u90, u91, u92 make up a 32-bit data bus and the data values are to be scanned into these devices. Define a signal to manipulate the entire 32-bit bus instead of masking and shifting the data into the individual devices. The signal would look like this:

```
SIGNAL data_bus u89.output(7,0), u90.output(7,0),
        u91.output(7,0), u92.output(7,0);
```

Illegal Constraint Declarations

The power inherent in controlling all scannable nodes in a board or module is not without risk. Conflicts can arise if the programmer is not careful when scanning into the devices (i.e. bus contention on a common bus). To prevent accidental conflicts, ASSET allows creation of a set of constraints for a scan board or module. Once defined, the constraints are always checked prior to any scan operation. If a constraint is violated by the execution of a scan operation, the operation is skipped and an error message displaying the relevant constraint is issued. Constraints consist of any valid C++ expression defined in terms of any scannable fields or signals that have already been defined. ASSET considers that a violation occurs if the expression evaluates to TRUE (non-zero). For example, in the GDM when the Floating Point Translator ASIC field OEZ is low and DIR is high, the ASIC will back-drive the data bus and damage the device. A constraint for this situation can be defined. The following expression defines the illegal state for the GDM:

```
ILLEGAL !u20.oez( ) && u20.dir( );
```

In this expression, the ! says to invert the value in u20.oez and logically AND it with the value in u20.dir. If this evaluates to TRUE, then an illegal state exists. The empty parenthesis indicates the entire field is used in the constraint. If a number inside the parenthesis is entered, only that particular bit in the field is used in the constraint. For more information on constraint checking in general, see the application article "Using the Constraint Checker."

Configuration Editor

The Configuration Editor is a software program that allows creation of ASSET configuration files. It is a context-sensitive text editor that prevents coding errors by enforcing correct syntactical structure. It also gives information about the items in the configuration file that it is currently working on. For a complete tutorial on the use of the Configuration Editor, refer to the ASSET User's Guide.

Configuration Translator

The Configuration Translator is a program that will take device or board or module configuration files and translate them into the C++ programming language. While translating, the translator checks the syntactical structure of the configuration file and displays an error message when a problem is detected. After the translation, two files are generated that have the following extensions: .hpp (header file) and .cpp (the configuration description). These files may then be compiled to produce an executable program. Additional code may be added using a stand-alone text editor. For more information on the Configuration Translator, refer to the ASSET User's Guide.

Conclusions

Configuration files provide an easy means of describing the configuration of a device or board/module to ASSET. ASSET configuration files also provide the ability to define signals to aid the programmer's ability to specify constraints to prevent accidental damage to devices or board/modules. By using the Configuration Editor, configuration files can be quickly developed without the burden of remembering the syntactic structure of configuration files. The Configuration Translator can be used to automatically generate ASSET code to be compiled and linked.

Built-In Self-Test (BIST) Using Boundary Scan

by Lee Whetsel

This paper was presented at the ATE West and Instrumentation Conference East, 1989.

Abstract

The IEEE standard boundary scan framework and four-wire serial testability bus will have a positive impact on design for testability at all levels of electronic assembly. While boundary scan and the four-wire test bus are not going to solve all the testing problems facing the electronics industry, they are significant steps in the right direction. They also form the basis from which other test techniques can be developed to further facilitate the testing of chips and systems. This paper describes a test architecture, based on the IEEE 1149.1 boundary scan and test bus standard, that extends the capability of boundary testing from a purely scan-based structure into one that also supports a built-in self-test (BIST) capability.

Introduction

Before the formation of the Joint Test Action Group (JTAG), the founding fathers of IEEE 1149.1 standard, the Test Automation Department of TI's Defense Systems & Electronics Group (DSEG) had studied boundary scan as a potential method to improve the test, integration, and maintenance of systems being designed for the Department of Defense (DoD). From this study, an architecture was developed, along with a library of specialized test cells particularly well-suited for boundary scan and BIST applications.

SCOPE Architecture

The test architecture, referred to as System Controllability, Observability, and Partitioning Environment (SCOPE), provides boundary scan and BIST capability to each input and output pin of the host IC. The architecture is supported by a library of modular bit slice testability cells (SCOPE cells) that offer a range of boundary test capability. Some of the cells are targeted for simple boundary scan applications, while others support the design of more sophisticated boundary test circuits such as pseudorandom and binary pattern generators and parallel signature analysis registers. During test, the SCOPE cells receive control from the test bus interface to execute a boundary scan or BIST controllability and observability test operation. One novel feature of the SCOPE architecture is its ability to activate the boundary test circuits while the host IC is in a normal operational mode. This capability provides boundary test features to support system integration, emulation, and at-speed testing.

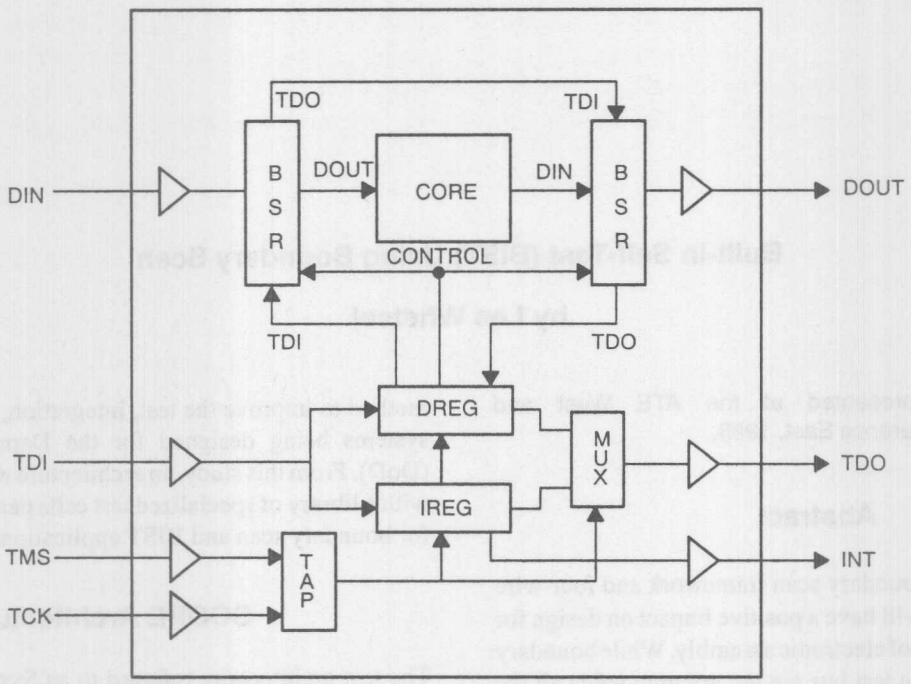


Figure 1. IEEE 1149.1 Architecture

Adapting SCOPE to 1149.1

Adapting the SCOPE architecture to the 1149.1 standard was not a difficult task, since the existing test bus and architecture had similar functionality and required the same number of serial test bus pins. The following sections of this paper describe the boundary test features provided in the SCOPE architecture to support the mandatory test instructions required by the 1149.1 standard, as well as some of the internal test capabilities developed to extend the range of boundary testing.

1149.1 Architecture and Test Bus

In Figure 1, the 1149.1 boundary scan architecture and four-wire test bus interface is shown. The test architecture consists of a test access port (TAP), two separate shift register paths for data (DREG) and instruction (IREG), and a boundary scan path bordering the IC's input and output pins. The boundary scan path is one of two required scan paths residing in the DREG and is shown outside the DREG for clarity. The other required DREG scan path is a bypass register that consists of a single scan cell and is used to provide an abbreviated path through the DREG when testing is not being performed. In addition to these two required data registers, any number of user-defined shift registers can be included in the DREG to allow expanding the architecture to support additional test capabilities.

The 1149.1 test bus interface consists of a test data input (TDI), a test data output (TDO), a test mode select (TMS), and a test clock (TCK). The TDI is routed to both the DREG and IREG and is used to transfer serial data into one of the two shift registers during a scan operation. The TDO is selected to output serial data from either the DREG or IREG during a scan operation. The TMS and TCK are control inputs to the TAP. These control signals direct the operation of the architecture to perform scan operations to either the DREG or IREG, or, if test operations are not being performed, to issue a reset condition to the test logic.

Interrupt Pin and Application

The interrupt output signal pin is not defined in the IEEE 1149.1 specification but has been envisioned as a potential mode of operation for SCOPE architectures. Although this signal is not a standard function in every SCOPE IC product, it is a powerful function that can be implemented when using TI's ASIC library SCOPE cells.

In Figure 1, the interrupt (INT) output signal issued from the IREG is a signal defined in the SCOPE architecture. This signal can be used for, among other things, the output of a parity

test indication of whether the instruction shifted into the IREG has the correct number of ones for even parity. The test instructions for the SCOPE architecture are all designed to a fixed length and include even parity. If an instruction is shifted in with odd parity, the interrupt output signal is set low when the TAP enters the pause-IR state to indicate the error back to a bus controller device. The pause-IR state is a steady state in the TAP controller's state diagram (see the IEEE 1149.1 Specifications) that can be entered before terminating the instruction shift operation. The SCOPE architecture takes advantage of the pause-IR state to verify that the instruction shifted in has the correct parity. If the INT output is low, indicating a parity error condition, the bus master will repeat the instruction scan operation again. If the INT output is high, indicating a good parity condition, the bus master will complete the instruction scan operation by updating the instruction into the IREG output latch to issue the appropriate test control to the architecture.

Test Bus and Interrupt Pin Connections

In the example circuit of Figure 2, three ICs are shown interconnected functionally via direct wiring bus paths. During normal operation, ICs 1, 2, and 3 receive input via their data input (DIN) buses and issue output via their data output (DOUT) buses to execute a designed circuit function. Also, the three ICs are shown interconnected in a serial ring configuration via the four-wire test bus. The TIC block in each IC stands for test interface and control and is representative of the TAP, IREG, and DREG sections shown in Figure 1. The 1149.1 test bus is designed to allow scan operations to occur through the TIC while the IC is operational. This is possible because the test pins and the required test logic of the boundary scan architecture (TAP, IREG, DREG boundary and bypass registers) are dedicated for test and cannot be reused for functional purposes.

The INT output signal from each IC is input to an external AND function to produce a global INT output signal. The global INT signal from the AND function is input to a bus master device, which includes a five-pin interface to support the four required 1149.1 test bus signals and the additional INT signal. During the pause-IR state of an instruction register scan operation, each local INT signal outputs a parity pass or fail condition. If all local INT outputs are high, the AND function outputs a high to the bus master to confirm good parity. If one or more local INT outputs are low, the AND function outputs a low to the bus master to indicate an instruction parity failure. The advantage of using an active AND function to combine the local INT outputs versus a passive wired-OR configuration is speed. While speed is not a high priority for global parity testing, other INT functions defined in the SCOPE architecture do require a minimum delay between the occurrence of one or more local INT outputs and a resulting global INT output signal to the bus master.

1149.1 Boundary Test Instructions

The following is a description of the required 1149.1 test instructions that are included in the SCOPE architecture. The other optional 1149.1 instruction (INTEST, IDCODE, RUNBIST) can be implemented as required.

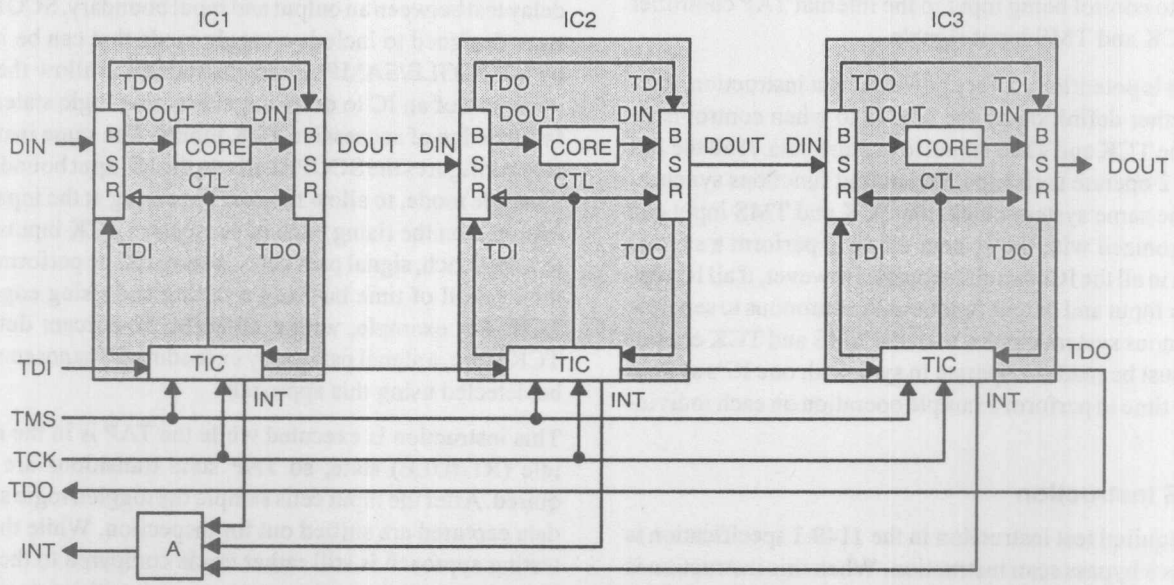


Figure 2. IEEE 1149.1 Ring Configuration

EXTEST Instruction

One of the required test instructions in the 1149.1 specification is defined as an external boundary test (EXTEST) instruction. When this instruction is shifted into the IREG of the ICs in Figure 2, the ICs are forced into an off-line test mode. While this instruction is in effect, the test bus can shift data through the boundary scan registers of each IC to observe and control the external DIN and DOUT buses, respectively. As seen in the circuit of Figure 2, serialized test patterns can be shifted in and applied from the output boundary scan register's DOUT to drive the wiring interconnect to the DIN of neighboring ICs. After the test pattern is output, the DIN of the receiving ICs are captured. Following the data capture operation, the boundary registers are shifted to load the next external test pattern and to extract the response from the first test pattern. This process is repeated until the wiring interconnects are tested. This test provides a simple and thorough verification of the wiring connections between ICs in a circuit and is the key motivation behind the JTAG initiative.

SAMPLE Instruction

A second required test instruction in the 1149.1 specification is defined as a boundary register sample instruction. When this instruction is shifted into the IREG of one or more of the ICs in Figure 2, the IC remains fully operational. While this instruction is in effect, a DREG scan operation will transfer serial data from the TDI, through the boundary scan register, to the TDO of the IC.

During this instruction, the data appearing at the DIN are captured and shifted out for inspection. The data are captured in response to control being input to the internal TAP controller via the TCK and TMS input signals.

While this is potentially a very powerful test instruction, it requires further definition by the user as to when control is issued on the TCK and TMS signal to capture data. If all the ICs in Figure 2 operate their input and output functions synchronous to the same system clock, the TCK and TMS input can be synchronized with the system clock to perform a sample operation in all the ICs simultaneously. However, if all ICs operate their input and output functions synchronous to separate asynchronous system clocks, then the TMS and TCK control signals must be made to operate in sync with one IC's system clock at a time to perform a sample operation on each individual IC.

BYPASS Instruction

A third required test instruction in the 1149.1 specification is defined as a bypass scan instruction. When this instruction is shifted into the IREG of one or more of the ICs in Figure 2, the IC remains fully operational. While this instruction is in effect, a DREG scan operation will transfer serial data from the TDI, through a single internal scan cell, to the TDO of the IC. The purpose of the instruction is to abbreviate the scan path through ICs that are not being tested to only a single scan clock delay.

SCOPE Boundary Test Instructions

To expand the test capability of a boundary scan architecture to support a broader range of testing needs, additional test instructions and hardware capabilities are defined in the SCOPE architecture. Following is a description of some of these additional test instructions and the benefits gained from them.

TOGGLE/SAMPLE Instruction

While the 1149.1 EXTEST instruction is excellent for testing the integrity of wiring interconnects between the ICs, it cannot be used effectively to test for timing delay problems that may occur between an output buffer of a driving IC and an input buffer of a receiving IC. To test for signal path time delays, data must be applied from the outputs of one IC and be sampled into the inputs of another IC in a short amount of time. In attempting to perform a delay test by transitioning through the TAP controller's state diagrams, it would take at minimum 1.5 TCK cycles from the time data are applied from one IC's output boundary in the update-DR state to when the data could be captured at the input boundary of another IC during the capture-DR state.

To provide a faster method of performing a simple signal path delay test between an output and input boundary, SCOPE cells were designed to include a toggle mode that can be invoked by a TOGGLE/SAMPLE test instruction to allow the output boundary of an IC to drive out alternating logic states on the falling edge of successive TCK inputs. The same instruction also configures the SCOPE cells on the IC input boundary into a sample mode, to allow the data appearing at the inputs to be captured on the rising edge of successive TCK inputs. Using this approach, signal path delay testing can be performed over the interval of time between a falling and rising edge of the TCK. For example, with a 10-MHz, 50-percent duty-cycle TCK input, a signal path delay exceeding 50 nanoseconds can be detected using this approach.

This instruction is executed while the TAP is in the run test/idle (RT/IDLE) state, so TAP state transitions are not required. After the input cells sample the toggled logic state, the data captured are shifted out for inspection. While this delay testing approach is still rather crude compared to the resolution of state-of-the-art test equipment, it still can serve a pur-

pose at the circuit board level. Some of the typical types of signal path delays that this technique can be used to identify are shown in Figure 3 and include combinational logic delays between the boundaries of larger ICs, delays associated with open collector-type output buffers, and delays associated with output buffers with high fanout loads.

TRISTATE AND BYPASS Instruction

When this instruction is shifted into the instruction register, the IC is set in an off-line test mode, and the output buffers are placed in a high-impedance state. While this instruction is in effect, the bypass register will be selected to shift data through the DREG from the TDI to the TDO. This instruction is included to ease in-circuit testing of ICs in a circuit that does not incorporate boundary scan. Since the output buffers can be set tristate, the concerns with backdriving the output buffers of one IC to apply a test to a neighboring IC are reduced.

SET AND BYPASS Instruction

When this instruction is shifted into the instruction register, the IC is set in an off-line test mode, and the input and output buffers are driven to a predetermined output pattern that has been scanned into the boundary register. While this instruction is in effect, the bypass register will be selected to shift data through the DREG from the TDI to the TDO. This instruction is defined to allow the output signals of one or more ICs in a circuit to be set and left in a preferred output state while testing is being performed on one or more neighboring ICs in the circuit. The key to this instruction is that it allows the bypass register to be selected while the IC remains in a test mode with outputs set as required for testing. The 1149.1 bypass instruction forces the IC into normal operation when the bypass register is selected; thus any preferred output signal setting for test cannot be maintained during the bypass instruction.

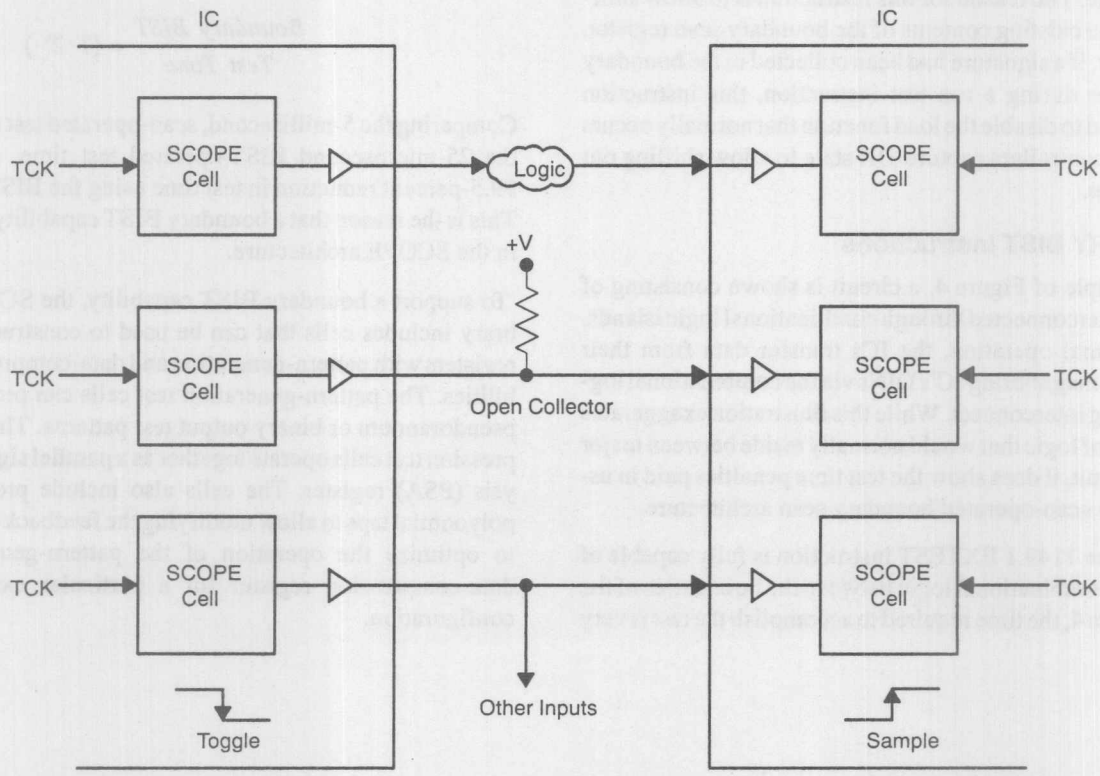


Figure 3. Using TOGGLE/SAMPLE to Test Path Delays

BOUNDARY SELF-TEST Instruction

When this instruction is shifted into the instruction register, the boundary scan cells are configured to allow self-testing of each cell in the scan path. The self-test exercises most of the logic in each test cell. The test involves initializing the cells with a test pattern via a scan operation, followed by a second scan operation to shift out the test results. If the cells pass the self-test, the pattern shifted out during the second scan operation will be the inverse of the first pattern shifted in. During this instruction, the IC remains in its normal operation mode to allow self-testing to occur in the background, if desired.

BOUNDARY READ Instruction

When this instruction is shifted into the instruction register, the boundary scan register is selected to shift data from the TDI to the TDO. This instruction differs from the 1149.1 EXTEST and SAMPLE instruction in that the boundary cells remain in their present state during the TAP controllers capture-DR state. The reason for this instruction is to allow shifting out of the existing contents of the boundary scan register. For instance, if a signature had been collected in the boundary scan register during a run test instruction, this instruction could be used to disable the load function that normally occurs in the TAP controllers capture-DR state to allow shifting out the signature.

BOUNDARY BIST Instructions

In the example of Figure 4, a circuit is shown consisting of three ICs interconnected through combinational logic islands. During normal operation, the ICs transfer data from their DOUTs to a neighboring IC's DINs via the combinational logic and wiring interconnect. While this illustration exaggerates the amount of logic that would normally reside between major ICs in a circuit, it does show the test time penalties paid in using a purely scan-operated boundary scan architecture.

Although the 1149.1 EXTEST instruction is fully capable of testing the combinational logic between the boundaries of the ICs in Figure 4, the time required to accomplish the test is very long compared to a BIST approach. For example, if the number of TCK inputs required to shift the boundary scan paths of the ICs to apply one test pattern is equal to (M), the TCK period is equal to (T), and the combinational logic island being tested has (n) inputs, the total boundary scan test time to apply an exhaustive test can be approximated by Equation (1). For a 10-MHz TCK, a 200-TCK scan operation, and an 8-bit-wide combinational logic island, the test time is equal to around 5 milliseconds.

$$\frac{Boundary\ Scan}{Test\ Time} = (T\ M\ 2^n) \qquad (1)$$

By eliminating the need to traverse the boundary scan paths of the ICs each time a new test pattern is to be applied, the M variable drops out of Equation (1) to produce the boundary BIST time in Equation (2). Using Equation (2), the time to test the same combinational logic island at the same TCK rate is equal to around 25 microseconds.

$$\frac{Boundary\ BIST}{Test\ Time} = (T\ 2^n) \qquad (2)$$

Comparing the 5-millisecond, scan-operated test time against the 25-microsecond BIST-operated test time, results in a 99.5-percent reduction in test time using the BIST approach. This is the reason that a boundary BIST capability is included in the SCOPE architecture.

To support a boundary BIST capability, the SCOPE cell library includes cells that can be used to construct boundary registers with pattern-generation and data-compression capabilities. The pattern-generation test cells can produce either pseudorandom or binary output test patterns. The data-compression test cells operate together as a parallel signature analysis (PSA) register. The cells also include programmable polynomial taps to allow modifying the feedback connections to optimize the operation of the pattern-generation and data-compression register for a particular external logic configuration.

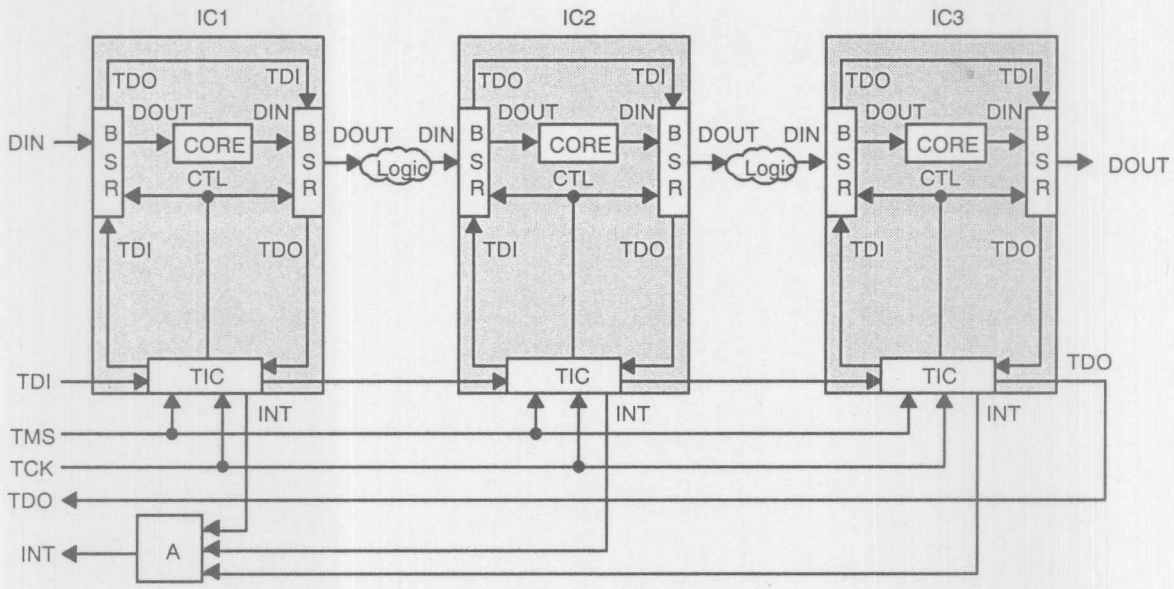


Figure 4. Testing Combinational Logic with IEEE 1149.1/BIST

When the boundary scan paths of the ICs in Figure 4 are designed using these advanced test cells, the combinational logic islands are tested much more easily and quickly. The boundary BIST test is set up by scanning seed values into the input and output boundary register sections that border the combination logic islands. After the boundary registers are set up, a run test instruction is shifted into the instruction register, and the test begins when the TAP controller enters its RT/IDLE state. During the test, the output patterns are applied from an ICs output boundary register on the falling edge of the TCK to drive the combinational logic. The output response from the combinational logic is input to the ICs input boundary register, where it is sampled into the PSA register on the rising edge of the TCK input. When the test is complete, the TAP controller transitions from the RT/IDLE state and shifts a boundary read instruction into the instruction register. The boundary read instruction selects the boundary register to allow the signatures collected to be shifted out for inspection.

Summary

The 1149.1 standard provides a basic framework for boundary testing. A user may take this basic framework and use it as it is or build on top of it the types of test structures and operations required to satisfy particular testing needs. This paper illustrates how an existing architecture can be modified to conform to the 1149.1 architecture. Also, adapting to the new architecture does not require giving up any pre-existing test capabilities. In addition, a boundary BIST approach was described and compared to a purely scan-operated boundary test approach. This comparison shows the benefits of a boundary BIST approach to be a significant reduction in the time to test combinational logic residing between the boundaries of ICs in a circuit.

References

[1] *Joint Test Action Group Specification*, Version 2.0. (Available from the author).

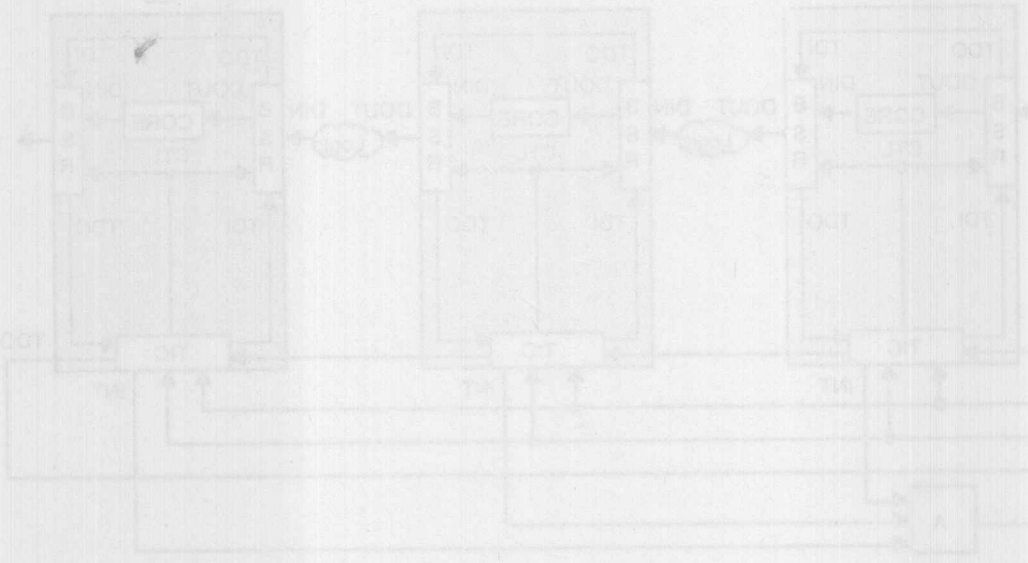
[2] *IEEE P1149.1, Draft 3, Proposal Specification.* (For information on obtaining a copy, contact the author.)

[3] C. Maunder and F. Beenker. *Boundary Scan: A Framework for Structured Design-for-Test*, Proceedings IEEE International Test Conference, 1987, pp. 714–723.

[4] F. Beenker. *Systematic and Structured Methods for Digital Board Testing*, Proceedings IEEE International Test Conference, 1985, pp. 380–385.

[5] M.M. Pradham, R.E. Tulloss, F.P.M. Beenker, and H. Bleeker. *Developing a Standard for Boundary Scan Implementation*, Proceedings International Conference on Computer Design, 1987, pp. 462–466.

[6] L. Whetsel. *A Proposed Standard Test Bus and Boundary Scan Architecture*, Proceedings International Conference on Computer Design, 1988, pp. 330–333.



Counting Techniques with SCOPE

by Adam Cron

Abstract

This article describes the use of Texas Instruments ASIC Library SCOPE Cells in the construction of built-in-test structures that generate a sequential pattern of numbers. These counting registers are accessed via the IEEE 1149.1 scan interface of an ASIC, and are compatible with the IEEE 1149.1 specifications and standards.

Objective

This application note is intended to give a design engineer an example circuit for implementing counting registers using Texas Instruments 1-μm Standard Cell ASIC Library SCOPE components.

Introduction

Test Register Uses

The counting test register is useful in Built-In-Test (BIT) circuits surrounding RAM, ROM, or other combinational logic blocks. (Refer to Figure 1 for an example illustration.) A sequential set of numbers can generated by the test register to address a memory element, while another test register generates random data to be written to the memory element. After the memory element is full of data, the element may then be sequentially read from while a signature is taken of the data stream from the element. The implementation of the counting test register described here is designed with test logic compatible with the IEEE 1149.1 standard.

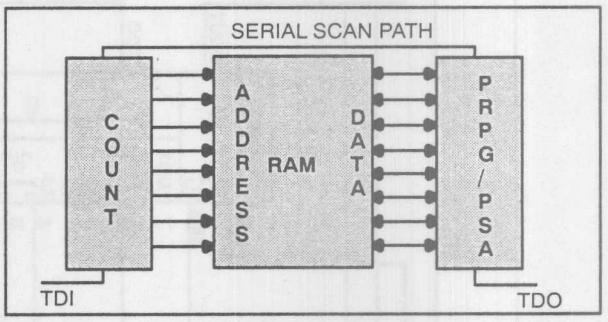


Figure 1. Testing with Counting and PRPG/PSA

Access to Counting Registers

The counting test registers should be accessible via the IEEE 1149.1 scan interface to the device using these registers. The scan bus would be used to load the registers with the starting value of the sequential pattern set, then the test clock (or system clock for some applications) would clock the register to generate the test patterns.

Benefits

- The test host need only store starting values for the count sequence (which may all be zero) and the number of clocks necessary to run the test.
- Exhaustive test vectors do not have to be stored as for normal boundary scan testing techniques.
- To test a target device in a system, the test time will usually be drastically reduced due to the fact that individual test patterns are not being scanned to the target (through other ICs), but only a few scans are done to the target. The test can then proceed at the test clock rate, with counting patterns being generated on each clock cycle.

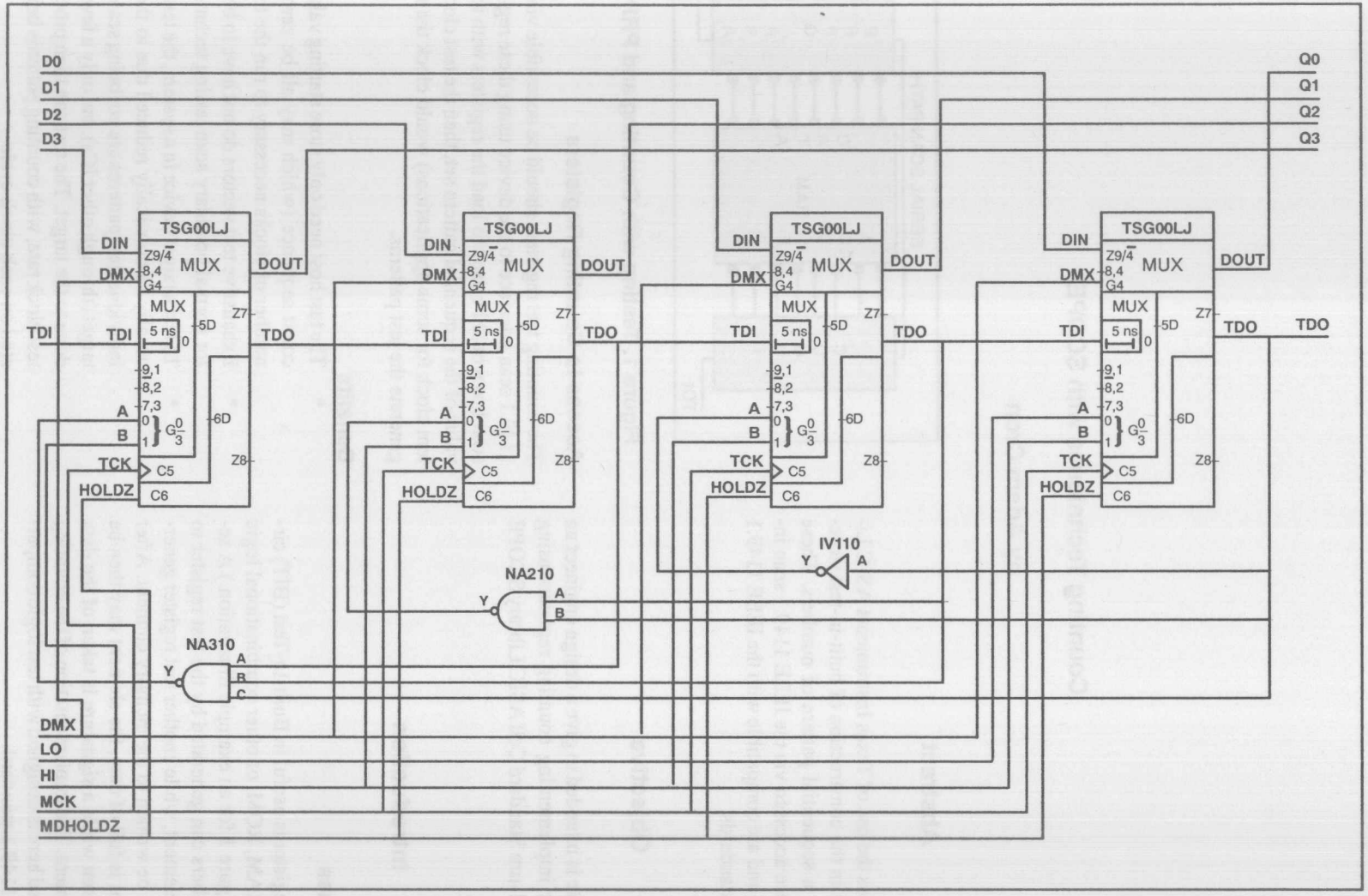


Figure 2. Count-Up Register

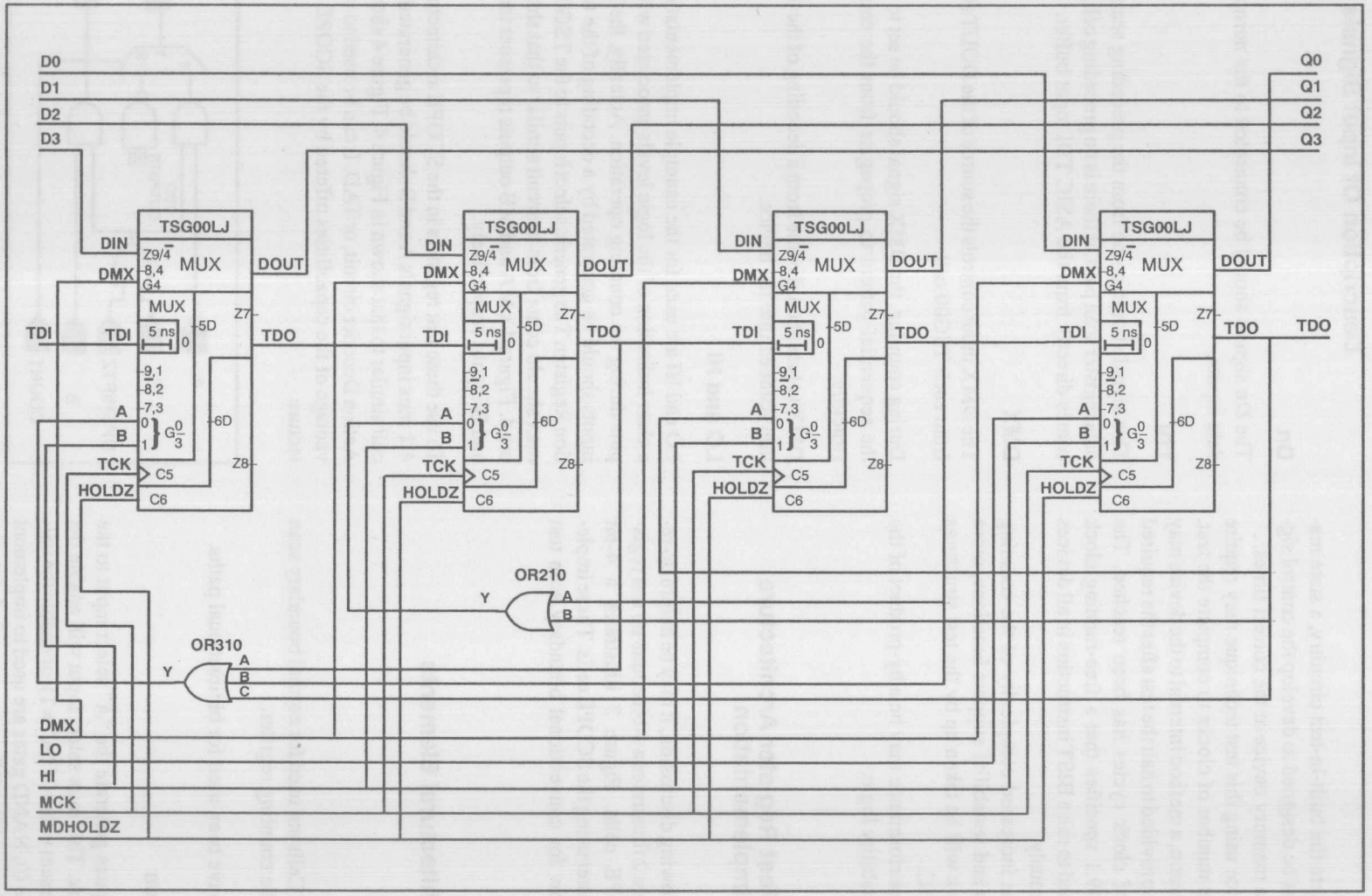


Figure 3. Count-Down Register

Penalties

1. To complete the built-in-test circuitry, a state machine must be designed to develop the control signals to the memory device at the correct times.
2. Each device using this test technique may require a different number of clocks to complete the test. For this reason, a method internal to the device may have to be provided to halt the test after the required number of clock cycles has been reached. The IEEE 1149.1 specifies that a free-running clock may be used to run a BIST instruction in all devices simultaneously.
3. Due to the increased complexity of the counting structures and testability support functions, more silicon area will be taken up by the test structures in the ASIC.
4. Reduced performance may be a by-product of the added testability logic.

Counting Test Register Architecture Implementation

(Throughout the following discussion, it may be helpful to refer to Figure 8.) Figure 2 illustrates a 4-bit count-up test register using the SCOPE cells. Figure 3 illustrates a 4-bit count-down test register using the SCOPE cells. These implementations also allow for conventional boundary scan test techniques.

Architectural Elements

TSG00

The TSG00 SCOPE Cells are used for normal boundary scan techniques and for the counting register.

TSB00 cells could have been used for bidirectional paths.

NAND or OR Gates

The NAND or OR gates generate the "A" select input to the TSG00 4:1 input mux. This mux select input will enable the TSG00 to hold its present value (if "A" = 1), or enable the value to toggle (if "A" = 0). NAND gates are used to implement the count-up function, while OR gates are used for the count-down function.

Description Of Input Signals

Dn

The Dn signals should be connected to the normal (system) data inputs.

TDI

This signal should come from the preceding scan cell in the data register scan path. If there is no preceding cell, this signal comes directly from the ASIC TDI input buffer.

DMX

The DMX input controls the source of the DOUT (Qn) signals from each TSG00 cell.

During counting, the DMX signal should be set to 1 to enable the sequential pattern to propagate from the count register structure.

DMX values should come from a decoding of the test instruction scanned into the device.

LO and HI

LO and HI are used for the example implementations to give a clear indication of the logic levels associated with these inputs during the counting operation. Actually, the LO and HI inputs should be generated by a decoding of the test Instruction Register. The generic select inputs to the TSG00 input 4:1 mux should come from a circuit similar to that shown in Figure 4. Figure 4's MA and MB outputs represent the TSG00 A and B inputs, respectively.

To use these test registers in the SCOPE architecture, TSG00 4:1 mux input signals A and B should be generated from a circuit similar to that shown in Figure 4. Figure 4 shows the Test Action Decoder circuit, or TAD. It can be used to take full advantage of the capabilities offered by the SCOPE Cell architecture.

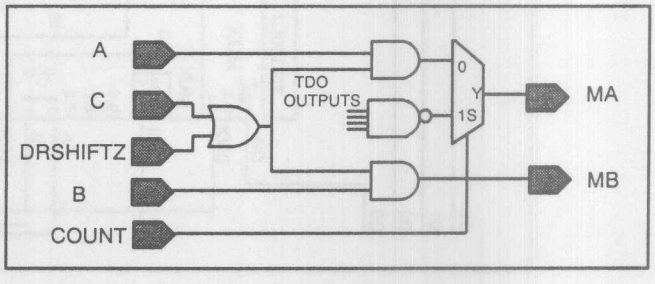


Figure 4. Test Action Decoder with Count Select

A, B, C, and COUNT come from a decoding of the test instruction register. Signal A and B control the function of the SCOPE Cell during the Data Register Capture state of the Test Access Port (TAP). C (in conjunction with A and B) controls the operation of the SCOPE Cells during the Data Register Shift state of the TAP. The DRSHIFTZ signal comes from Texas Instruments' TAP (TS002). The COUNT signal also comes from a decoding of the Test Instruction Register. When 0, the A, B, and C signals control the function of the test register. When COUNT is 1, MA is controlled by the TDO outputs of the preceding TSG00 cells.

The diagrams in Figure 5 illustrate the function of the SCOPE Cells for the two logic values of C (while COUNT is 0).

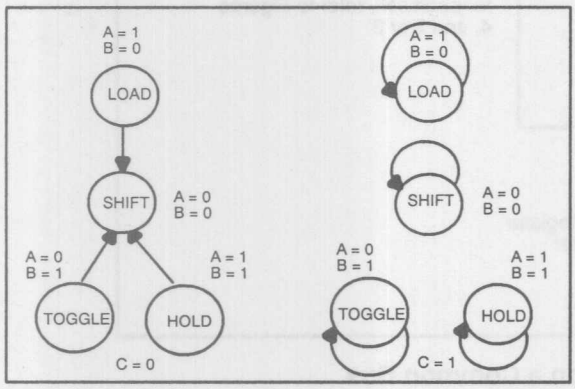


Figure 5. Actions Taken in Capture-DR and Shift-DR States of the Test Access Port

For counting operations, B and C should be set to 1, and COUNT should be set to 1. This configures the SCOPE Cells to always count (although, an individual cell may toggle its cell value or keep the same value on any given MCK (TCK) cycle, due to the value of A to the cell).

MCK

The source of MCK changes with the COUNT signal. During data shift operations, MCK should be provided by DRCK from the TS002. During count operations, MCK should come from TCK while the TS002 is in the Idle/Run Test state.

The IDLE signal is an output of the TS002. It is active (high) when the TS002 is in the Idle/Run Test state. Figure 6 illustrates this clock selection scheme.

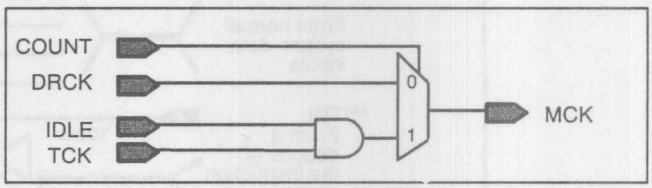


Figure 6. Test Clock Selection and Gating

MDRHOLDZ

MDRHOLDZ should follow the DRHOLDZ signal from the TS002 during normal scan operations. During count operations, MDRHOLDZ should follow the inverse of TCK ($\overline{TCK}$). Figure 7 illustrates a circuit that can be used to generate the MDRHOLDZ signal.

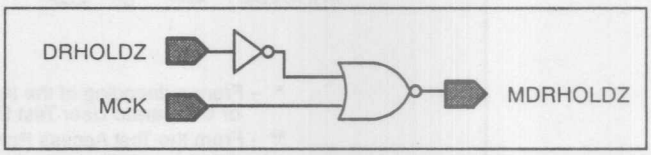


Figure 7. Circuit for Use with Count Operations

Description of Output Signals

Qn

The Qn data outputs from the TSG00 cells follow the functional (normal) Dn inputs when the count register is configured in its normal transparent mode (when DMX is low or 0). The Qn data outputs can be held to a specific scanned-in value or generate counting patterns if commanded to by the test instruction. The Qn outputs should be connected to the normal device outputs or to the device core inputs.

TDO

TDO should be connected to the TDI input of another scan register, or to a data register multiplexer input on its way to the TDO output of the ASIC.

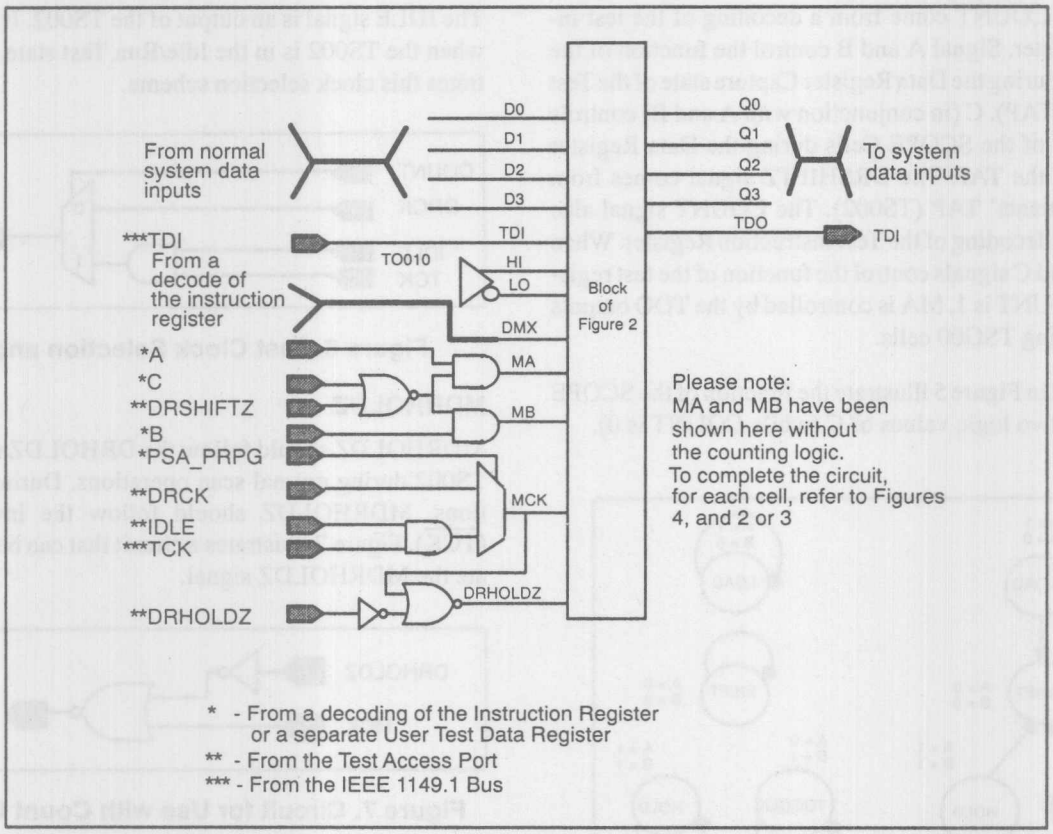


Figure 8. Register Outputs on a Common Bus

Practical Application Notes

Trade-offs

Due to the extra circuitry involved to implement a counting register, a designer should trade off this test solution against the simpler (smaller) boundary scan solution.

Test Operation

The counting operation should occur in the RUN-TEST/IDLE state of the Test Access Port according to the IEEE 1149.1 specification. The circuit should stop counting when not in that state.

Test Sequence

To control this test, first pre-load the register with a starting value using an instruction that selects this register into the scan path. Next, load in the BIST instruction or other test instruction into the instruction register. After the test is complete, scan out or observe the BIST or test results.

Designing Application-Specific Integrated Circuits (ASICs) with Boundary Scan Logic

by Steve Sparks

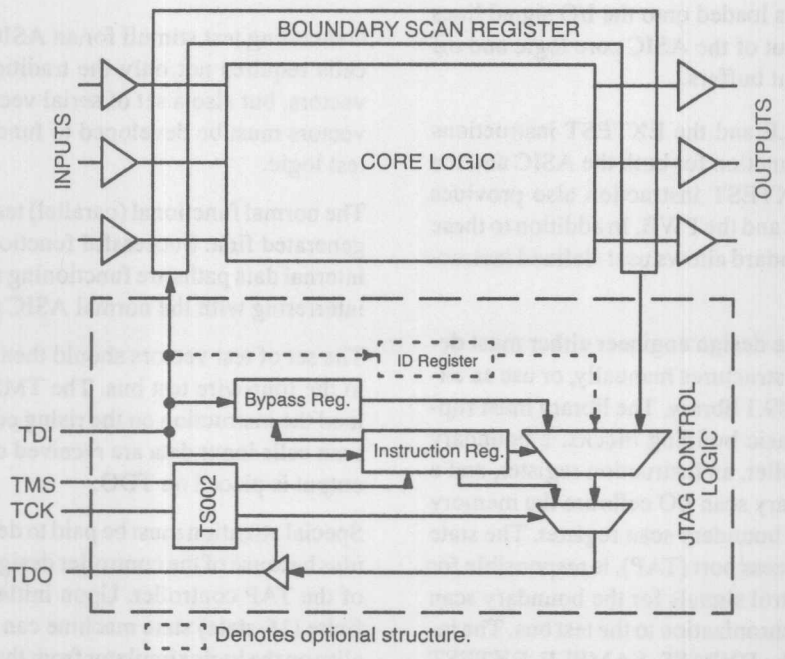
This paper was presented at the Government Microcircuit Applications Conference (GOMAC), 1989.

Introduction

The cost of testing high-end electronics is forcing manufacturers to start designing their products for testability. Also, MIL-STD-2165 requires military contractors to design-in testability. The Institute of Electrical and Electronics Engineers (IEEE) 1149.1 Boundary Scan Specification is being adopted rapidly by both military and commercial vendors as a way to implement component and board-level testability.

The IEEE Joint Test Action Group (JTAG) 1149.1 standard defines a boundary scan architecture and methodology for testing printed wiring boards (PWBs) where complex integrated circuits (ICs) and high-density packaging (such as surface-mount technology) are used. The boundary scan architecture also can be applied to ASIC devices. In addition to providing PWB-level test capability, the boundary scan logic also can be used to help detect errors in the ASIC itself. Figure 1 shows an ASIC containing the boundary scan architecture.

When incorporating the boundary scan architecture into an ASIC, there are several issues that must be addressed. This paper describes a chronological design flow addressing each of these issues beginning with logic design and capture considerations, simulation and debug practices, ASIC tester methodology, and test pattern development.



(TI's Boundary Control Register is an example of a user-defined test register.)

Figure 1. ASIC Incorporating IEEE 1149.1 Boundary Scan Architecture

Logic Design

The decision to incorporate boundary scan architecture in an ASIC first should be considered from a board- or system-level point of view. If boundary scan is to be used as a system requirement, then it is very important to define the instructions necessary at the system or board level first, and then implement the appropriate instructions on the ASIC. As a minimum, the IEEE 1149.1 standard dictates that the ASIC must be able to perform three instructions: BYPASS, SAMPLE, and EXTEST.

The BYPASS instruction routes the test bus input directly to the next 1149.1 device, and "bypasses" the boundary scan register of the ASIC. This instruction does not affect ASIC operation.

The SAMPLE instruction has the boundary scan cells take a "snapshot" of the logic values on the ASIC input/output (I/O) cells and serially shifts these data out for examination. ASIC operation is unaffected by execution of this instruction.

The EXTEST instruction interrupts normal ASIC operation, and control of the ASIC is given to the four-wire IEEE 1149.1 test bus. During this instruction, a snapshot is taken of the logic values on the ASIC I/O cells and routed out onto the test bus. As these data are being shifted out serially, the boundary scan cells are being reloaded with new data (normal test stimuli). This new test stimuli then is loaded onto the I/O signal lines and allowed to propagate out of the ASIC core logic and off the ASIC through the output buffers.

Note that both the SAMPLE and the EXTEST instructions provide the observability function for both the ASIC and the surrounding PWB. The EXTEST instruction also provides controllability for the ASIC and the PWB. In addition to these instructions, the 1149.1 standard allows user-defined instructions.

To start the logic design, the design engineer either must design the boundary scan test structures manually, or use an existing vendor-generated 1149.1 library. The library must support a minimum of four basic building blocks: a boundary scan I/O cell, a state controller, an instruction register, and a bypass register. The boundary scan I/O cells are the memory elements that comprise the boundary scan register. The state controller, called the test access port (TAP), is responsible for generating the 1149.1 control signals for the boundary scan cells and providing the synchronization to the test bus. The instruction register decodes the BYPASS, SAMPLE, EXTEST and any additional user-defined instructions and provides the control signals for the ASIC test logic. The bypass register implements the BYPASS instruction. Additionally, device identification register and test data register are optional structures that are used to extend the basic functionality of the 1149.1 specification. Texas Instruments offers a predefined library of 1.2-μm complementary metal-oxide semiconductor (CMOS) technology, 1149.1-compatible cells called System Controllability, Observability, and Partitioning Environment (SCOPE).

The IEEE 1149.1 standard contains a number of design requirements that must be followed to ensure that the ASIC works properly with other 1149.1 devices. First, a boundary scan cell must be placed at each ASIC I/O cell except the four test bus signals: test mode select (TMS), test clock (TCK), test data in (TDI), and test data out (TDO). Second, the TMS and TDI test bus signals must use a pullup cell to prevent unstable TAP controller operation. Third, the optional reset signal to the TAP controller must not be initializable from the system reset signal, since this would limit debug capabilities. Also, if this reset signal is not an ASIC I/O pin, then an on-chip power-up clear cell should be used to terminate this signal to reduce the risk of system I/O three-state signal contention during power-up. These requirements represent the most common implementation guidelines, and additional rules can be obtained from the actual specification.

Simulation

Generating test stimuli for an ASIC that uses boundary scan cells requires not only the traditional parallel or broadside vectors, but also a set of serial vectors. This set of serialized vectors must be developed to functionally verify the 1149.1 test logic.

The normal functional (parallel) test vectors should always be generated first. Successful functional vectors verify that the internal data paths are functioning and that the test logic is not interfering with the normal ASIC operation.

The set of test vectors should then be developed and applied to the four-wire test bus. The TMS signal is used to serially load the instruction on the rising edge of TCK. The boundary scan cells input data are received on TDI, and the scan chain output is placed on TDO.

Special attention must be paid to development of the test stimulus because of the controller design and user implementation of the TAP controller. Upon initialization, this synchronous finite (16-state) state machine can exhibit different functionality on the logic simulator from that of the actual ASIC at test. This simulation/hardware mismatch can be eliminated if the reset signal of the TAP controller is used properly.

To illustrate this condition, assume that the reset signal is used as an ASIC I/O pin and that it is biased to a logic "1" (or terminated through a pullup cell). At device power-up, the TAP controller of the actual ASIC will go to one of its 16 states (because all internal IC nodes either will be a logic "1" or "0"). The TAP can then be reset by either pulsing the reset pin (if it is an ASIC I/O pin) with a logic "0," or by setting the TMS input to a logic "1" for five TCK cycles. Unfortunately, logic simulators initialize internal nodes to an "unknown" at time "0." Hence, the TAP simulation model cannot initialize to one of the 16 states. Because of the logical feedback design of the state controller, a transition to one of the 16 states using the logic simulator can occur only by either pulsing the reset signal with a logic "0" (if it is an ASIC I/O pin) or by using the simulator FORCE command to initialize the TAP controller to any one of the 16 states.

A recommended method for initializing the TAP controller is to terminate the reset signal with a power-up clear cell. This implementation will force the TAP controller in both the logic simulator and the actual tester environment into the reset state shortly after time "0," or power-up. TMS then should be held to a logic "1" for five TCK cycles during device initialization. Improper initialization of the TAP controller not only will cause TAP controller problems, but also will cause the ASIC to malfunction because the boundary scan cells may not allow proper ASIC operation.

ASIC Tester Methodologies

After fabrication of the IC, it must be tested to ensure conformance to the design specification. For boundary scan designs, the ASIC tester must apply both normal functional and test vectors to the ASIC. Adherence to the above mentioned logic design and simulation sections should ensure successful tester results.

In reality, the only difference between normal functional test vectors and functional vectors might lie in the rate of application. The normal functional vectors are executed according to the operating frequency stated in the ASIC specification, but the test logic operating frequency is limited by the implementation of the boundary scan cells. The serialization of test stimulus and the lower operating frequency of the test logic requires more test time than equivalent normal functionality and performance tests.

Test Pattern Development

Test vector development for the IEEE 1149.1 test bus, if not properly managed, can become cumbersome because of the need to change from thinking about normal functional (parallel) vector generation to serial vector generation. The pattern development can be simplified by viewing instructions like microprocessor instructions. That is, each instruction may have both fixed and variable length elements. By properly partitioning an instruction, it is a relatively simple matter to construct a stimulus file that will generate the correct sequence of "1's" and "0s." These instructions (simulation stimulus files) then can be used like subroutines that can be assembled to form completed test programs.

First, the test patterns must be constructed to support only those instructions that were defined in the logic design phase of the ASIC development. Each test instruction consists of an instruction phase followed by a data phase. Test patterns are generated to represent each instruction by defining the instruction and data phase for each. The instruction execution is initiated by serially loading an instruction followed by the clocking of data into and out of the boundary scan register. The test patterns necessary to load each instruction are fixed because of the fixed bit length of the instruction register. The loading/unloading of data, however, may be of variable length, based on which the instruction is executed.

Several methods may be used to generate the actual test vectors. The first method would be to build vectors from the existing functional vectors. Figure 2 shows how functional vectors were modified to perform an 1149.1 SAMPLE instruction.

These vectors load a 3-bit instruction register with the SAMPLE instruction. Note that eight TCK cycles are required to load this instruction.

Figure 3 shows that a minimum of six vectors, or TCK cycles, is required to simultaneously load and unload the boundary scan register data during a SAMPLE instruction. The actual number of cycles needed will depend on the number of boundary scan cells in the scan chain.

These vectors illustrate the loading/unloading of boundary scan register data. Note that the number of data cycles requires [6 + (4 + N)] TCK cycles, where "N" is the number of unidirectional and bidirectional I/O cells in the boundary scan register.

Because an instruction consists of an instruction phase followed by a data phase, serial test vectors can be generated by concatenating these instruction/data sequences to perform each instruction.

```
                    s s s s s s
                    i i i i i i T T T T
                    g g g g g g M C D D
                    1 2 3 4 5 6 S K I O
SETR P : =T ' x . . . . x H C L Z;
SETR P : =T ' x . . . . x H C L Z;
SETR P : =T ' x . . . . x H C L M;
SETR P : =T ' x . . . . x H C L 1;
SETR P : =T ' x . . . . x H C L 0;
SETR P : =T ' x . . . . x H C L 0;
SETR P : =T ' x . . . . x H C L 1;
SETR P : =T ' x . . . . x H C L M;
```

ASIC/IO device pins

x . . . . = ASIC I/O data which is defined by the programmer. (or Don't Care)

H,L = Logic '1' and '0' inputs

1,0 = Logic '1' and '0' outputs

M = mask to ignore signal transitions

Figure 2. Instruction Sequence of the IEEE 1149.1 SAMPLE Instruction

```
                    s s s s s s
                    i i i i i i T T T T
                    g g g g g g M C D D
                    1 2 3 4 5 6 S K I O
SETR P : =T ' x . . . . x H C i Z;
SETR P : =T ' x . . . . x H C i M;
SETR P : =T ' x . . . . x H C i o;
SETR P : =T ' x . . . . x H C i o;
SETR P : =T ' x . . . . x H C i M;
SETR P : =T ' x . . . . x H C i Z;
```

i = For TDI represents required input to the scan chain.

o = For TDO represents data unloaded from scan chain.

Figure 3. Data Sequence of the IEEE 1149.1 SAMPLE Instruction

A second, and more efficient method of creating functional 1149.1 test vectors is to use a programming language like Pascal. This requires the generation of a program in which each of the instructions is represented as a procedure. Each procedure then would consist of an instruction phase followed by a data phase. From Figure 2, it should be noted that there are eight TCK cycles, with TMS and TDI being used as variables of a defined sequence, or logic value. The instruction phase, therefore, could be implemented by a simple loop. For the data phase (illustrated in Figure 3), the TDI's function has changed, and it must be represented as a parameter variable within the procedure. (The TMS' function is unchanged from the instruction phase.) The TDI value now represents data that are loaded into the boundary scan register. The data phase, therefore, also can be represented as a loop within the procedure, but with TDI being a variable.

Boundary Scan Benefits

The boundary scan architecture not only can help in isolating ASIC and PWB faults, but it also may assist as an error-correction mechanism. For example, assume that a system engineer has just discovered an incorrect result from an ASIC that is embedded in the system hardware. This system is still a prototype system when the "bug" is discovered. What would the system engineer do? If this ASIC did not incorporate boundary scan cells, either the system software would have to be modified to ignore this error condition or, if possible, the board in question would be pulled and a "blue-wire" installed to correct the problem. Both of these solutions would have undesirable side effects. (The software patch might require overlooking potential system malfunctions, and the "blue-wire" method in addition to reducing some of the functionality of

system, also might require the board to be pulled from the system and retested.) However, if boundary scan was implemented on the ASIC, it would be a simple matter for the system engineer to run the test program to the place at which the error would occur, serially scan-in the correct signal value(s), apply the change to the ASIC while the system is in a pause state, then allow the system to continue execution. The controllability feature of the IEEE 1149.1 test logic can be used as a "band-aid" to correct errors in an ASIC (or PWB) where other repair or replacement measures might be impractical.

Summary

The incorporation of boundary scan architecture within an ASIC design enhances the testability of the design. This testability must be designed into the ASIC using a top-down design approach. Although this additional test logic increases cycle time and cost during product development, this cost is minimal when compared with the expense required to test a complex system where testability was not included.

References

[1] *IEEE P1149.1 Specification*, Draft D3, 1989, IEEE.

[2] *SCOPE Cell Design Manual*, 1989, Texas Instruments Incorporated.

Designing a User Interface with ASSET

by Adam Sheppard

Introduction

Interfacing a user to a program is one of the biggest challenges when developing software today. It is no longer sufficient to supply the user with a working program. Experts say that a program with a bad user interface will not be used, regardless of how wonderful the program is. To aid in their quest for a better user interface, programmers are adopting the menu-and-window metaphor for their programs. ASSET includes a simple windowing library that allows programmers to create windows, write information to windows, and receive input from windows. This article introduces both the concepts of good user-interface design and the use of ASSET library functions to implement a sample user interface.

Designing a User Interface

A program's user interface includes all ways that the user and the program communicate interactively, including both user input and program output. Developing a good user interface for a program does not necessarily mean using menus and windows. A good user interface is one that adheres to certain design concepts that make a program easier to use and understand. These design concepts are simplicity, consistency, and clarity. They do not stand alone but rather complement each other. Improving a program in one of these categories will result in improvements in the other categories.

Simplifying the user interface of a program improves the user's comprehension of the program by making it easier to understand. Simplicity affects both program output and input. Cluttered screen displays make the important program output difficult to isolate. Setting aside specific areas of the screen for specific types of information results in a more readable display than writing the same information one line after another on a scrolling display. Similarly, complex command sequences, like those used in many of the most popular text editors, are both difficult to remember and difficult to associate with their related function. Using key sequences that are intuitive and mnemonic improves the user's retention of those sequences. Simplicity is a concern because the simpler a program is to use, the more likely a user will remember it.

Consistency of the program's user interface also affects how easily the user can remember its operation. A program should not only be consistent between its different sections, but also consistent with other programs the user is running. People can learn ten different sets of commands to operate ten different programs, but why should they bother? Many designers would rather force-fit a program to solve a problem it was never designed for rather than learn how to use a new and inconsistent program that solves their problem by design. For example, many people use Lotus 1-2-3 for their word processing simply because they already know how it works. Consistency applies to such things as organizing all menus in the same order, naming menu items the same that perform identical functions, using the same colors to display the same kinds of information, and recognizing the same key sequences to perform the same functions. This consideration is the main reason why graphical user interfaces are growing in popularity. The level of consistency between programs made possible by these operating systems far exceeds that of programs written on operating systems without a graphical user interface. When programs operate as identically as possible, users can focus more on the problem and less on the programs.

Clarity of the user interface involves careful choices in the wording and content of the information displayed. Wording and graphical content that violates the user's preconceived ideas must be avoided. The user needs to solve specific problems with a program. If it does not present the right information to solve those problems, it is inadequate. Likewise, the program cannot present too much information or the user will not be able to locate what is needed. Another problem is information that is too far removed from the problem being solved. If the user has to process the program output to get it in a useful

form, the program should be changed to incorporate that processing.

Users and programs communicate through the user interface. To make this communication as smooth as possible, programs should be written with principles of good user interface design in mind. A simple user interface improves comprehension and retention. A consistent user interface between programs removes the barriers that users face when switching between programs. A clear user interface provides the user with the correct information to solve the problem. Remember that users will embrace a program that meets their requirements and is simple to use, whereas they will reject a similar program that is difficult to use. The user interface can make or break a program.

A Sample Application

To illustrate the principles of good user-interface design and to demonstrate the ASSET window library functions, the shell of a test program, EXAMPLE, is developed in this paper. The sample program uses four windows – three for menus and one for data input. Features are added to the program as the library functions are introduced in the text. Before the program is developed, however, the format of the user interface must be determined.

The program EXAMPLE uses a menu and window interface, since this is the type of user interface supported by ASSET. Several conventions are followed in this program to satisfy simplicity and consistency requirements:

- Windows are used for all program output. The window currently being used for input and output has a double-line border. All other windows have single-line borders.
- Menus appear in the upper-left corner of the screen. Menu items are arranged vertically.
- When a menu item is selected, the window that opens as a result of the selection is offset down and to the right of the menu item. This allows the user to quickly see what menu items were chosen to reach the current command.
- When the selection of a menu item causes a submenu to appear, the name of that menu item is followed by ellipses (three periods). Menu items that cause functions to execute immediately (without further prompting) appears without ellipses.
- The user may exit a menu and return to the previous menu by pressing the escape (ESC) key. The program remains in a menu, executing commands, until the user presses the escape key to leave that menu.

The example program is the skeleton of a simple test program. It does no actual testing but has a user interface that could be found in a test program. The following outline establishes how the program functions and what its menus should look like:

Step 1. Begin Program

- Prompt for operator name
- Present main menu with the following items:
  - * Help, Tests, Debug, Exit
- Main menu, Help item:
  - * Display program help
- Main menu, Tests item:
  - * Present tests menu with the following items:

Step 2. Begin testing, First test, Last test

- * Tests menu, Begin testing item:

Step 3. Perform first through last test

- * Tests menu, First test item:

Step 4. Present which test menu with the following:

- Strap, Inst, Sample, Control, PSA/PRPG

Step 5. Set first test to perform based on menu choice

- * Tests menu, Last test item:

Step 6. Present which test menu (shown above)

Step 7. Set last test to perform based on menu choice

- * End menu
- Menu menu, Debug item:
  - * Invoke ASSET Run-Time debugger
- Main menu, Exit item:
  - * Exit program
- End menu

Step 8. End program

Figure 1. Example Program

Now that a program design is established, the program can be developed. Windows are the first addition to the program, because they are used as the basis for both menus and data entry. Next, menus are added to establish the command structure. And finally, data entry is added to receive some input from the user.

Establishing and Manipulating Windows

Window Concepts

A window is a rectangular area of the user's screen used to receive input from the user and display output from the program. A window can be thought of as a virtual terminal. Windows behave like terminals in that information written to a window never appears outside the window. A window has a cursor that shows where program input and output occurs next. When the window is filled, it scrolls to make room for more output, just like a terminal. Windows in ASSET have additional capabilities. Multiple, overlapping windows can be displayed on the screen. Each is entirely independent of the others. Any window can receive output at any time.

Creating a Window

In ASSET, a window must first be created before it can be displayed on the screen or receive any output. The function establish_window is used to create a window. It determines the size and initial position of the window on the screen. The following example creates a window:

```
WINDOW *win;
win = establish_window(20, 5, 8, 40);
```

Figure 2. Window Creation

Each window must first be declared as a pointer to a WINDOW structure. The function establish_window returns a pointer to a WINDOW structure, or NULL if the window could not be created because memory was exhausted. The first two parameters are the x and y coordinates of the window on the screen. The upper-left corner of the screen is at the coordinates x = 0, y = 0. The remaining two parameters are the height and width of the window in characters, respectively. This includes the window border, so the screen area inside the window that is available for output is actually two characters less in each direction (six lines high and 38 characters wide).

Displaying and Hiding Windows

Once a window is created, however, it is not automatically visible. The window is initially hidden so that it may be prepared before it is displayed. Preparing a window before displaying it makes the program appear visually cleaner and more responsive.

The function display_window is used to display an established window on the screen. It accepts the window pointer returned by establish_window as its only parameter. Likewise, the function hide_window is used to hide an established window. It accepts the same parameter type. The following example demonstrates the usage of these functions:

```
display_window (win);
/* Window is now visible */
      .
      .
      .
hide_window (win);
 /* Window is now hidden */
```

Figure 3. Window Visibility

When a window is displayed, it may overlap other windows already on the screen. If this occurs, the portions of the other windows that are obscured by the new window will be saved. If the new window is later hidden from view, the saved data will be restored to the screen, making it appear as if windows were just pieces of paper being laid on top of each other. Unlike paper, however, windows can be written to at any time, even if they are not at the top of the pile.

Window Attributes

A window has several attributes other than the position and size indicated by establish_window. The cursor determines the coordinates within the window where input or output occur next. The window cursor can be moved by calling the function wcursor. Its parameters include the window pointer of the window cursor to be moved and the x and y coordinates of the new position. Note that the cursor position is relative to the upper-left corner of the window, not the upper-left corner of the screen. This is in keeping with the idea that each window is a virtual terminal. It also prevents calls to wcursor from changing when the window is moved. The following example moves the window cursor to the coordinates x = 5, y = 6:

wcursor (win, 5, 6);

The window border created when establish_window is called is a single line. The function set_border is used to select be-

tween a single line, a double line, or a combination. Different border styles can be used to emphasize windows. The parameters to set_border include the window pointer and the new border style. The style table below shows the available border styles and their value:

Table 1. Window Border Styles

Code	Border Styles
0	Single-line border
1	Double-line border
2	Single-line horizontal, double-line vertical
3	Double-line horizontal, single-line vertical

The following example changes the window border to a double line:

```
set_border (win, 1);
```

Each window may have a title that is displayed centered in the top border of the window. Initially, the title is empty and is not displayed. The function set_title is used to change the title of a window. Its parameters include the window pointer and a character pointer to the title. Since the window does not make a copy of the title, the title pointer must be a constant string or a global or static character variable. The title may be made empty again by passing NULL for the title character pointer. The following example sets the title of a window:

```
set_title (win, "Edit File");
```

Normally, text written to the inside of a window appears in a single color. The function reverse_video is used to invert the color of all text that is subsequently written to the window. After all highlighted text is written, the function normal_video is used to restore text output to the normal color. Both functions accept the window pointer as their only parameter. The following example demonstrates the usage of these functions:

```
/* Output done now appears normal */
                    .
                    .
                    .
  reverse_video (win);
/* Output done now appears highlighted */
                    .
                    .
                    .
normal_video (win);
/* Output done now once again appears normal
*/
```

When a window is established, it uses a white foreground and a black background. The window colors may be changed by the function set_colors. The foreground and background colors may be changed for the window border, window title, normal_video text, and reverse_video text. The parameters include the window pointer, the window area to be affected, the background color, the foreground color, and the foreground intensity. Window areas include the following:

BORDER	Window border color
TITLE	Window title color
NORMAL	Window normal_video text
ACCENT	Window reverse_video text
ALL	All window areas (BORDER, TITLE, NORMAL and ACCENT)

Colors can be chosen from the following list:

RED	GREEN
BLUE	YELLOW
AQUA	MAGENTA
WHITE	BLACK

The foreground intensity can be either BRIGHT or DIM. Changes made by calling set_colors affect the entire window immediately, not just text that is output after the function is called. The following example sets the highlight color of a window to a red background and a high intensity white foreground:

```
set_colors (win, ACCENT, RED, WHITE, BRIGHT);
```

Window Output

When a window is established, it is empty. The program needs some way of writing text into the window. The function wprintf is used to print formatted text to a window. Its parameters include the window pointer to write to, a character pointer to a format specification, and other parameters as required by the format specification. As its name implies, wprintf is identical in function to the standard C function printf in operation, with the exception that output goes to a specific window instead of the screen. The format specification is identical (consult the Zortech library reference for information). The following example displays the value of an integer in a window:

```
int x = 5;
    wprintf (win, "The value of x is %d\n", x);
```

Destroying a Window

When a program is finished using a window, it should destroy the window to free both system memory and screen area. The function delete_window removes a window from the screen and frees the memory used by the window. Its parameters include the window pointer to delete. Once deleted, the window pointer cannot be used by any window function until it is ini-

tialized by another call to establish_window. If establish_window fails to create a window and returns NULL, that window should not be deleted (because it does not exist). The following example deletes a previously established window:

```
#include <stdio.h>
#include <stdlib.h>
#include <twindow.h>

void which_menu (void)
{
    WINDOW *which_win;
    which_win = establish_window (5, 7, 7, 17);
    set_border (which_win, 1);
    set_colors (which_win, ALL, BLACK, WHITE, BRIGHT);
    display_window (which_win);
    wprintf (which_win, " Strap test\n");
    wprintf (which_win, " Inst test\n");
    wprintf (which_win, " Sample test\n");
    wprintf (which_win, " Control test\n");
    wprintf (which_win, " Psa/prpg test\n");

    get_char();

    delete_window (which_win);
}
void test_menu (void)
{
    WINDOW *test_win;

    test_win = establish_window (3, 4, 5, 17);
    set_border (test_win, 1);
    set_colors (test_win, ALL, BLACK, WHITE, BRIGHT);

    display_window (test_win);
    wprintf (test_win, " Begin testing\n");
    wprintf (test_win, " First test...\n");
    wprintf (test_win, " Last test...");

    get_char ();
    set_border (test_win, 0);
    which_menu ();

    delete_window (test_win);
}

void main_menu (void)
{
    WINDOW *menu_win;

    menu_win = establish_window (1, 1, 6, 12);
    set_border (menu_win, 1);
    set_colors (menu_win, ALL, BLACK, WHITE, BRIGHT);

    display_window (menu_win);
    wprintf (menu_win, " Help\n");
    wprintf (menu_win, " Tests...\n");
    wprintf (menu_win, " Debug\n");
    wprintf (menu_win, " Exit");

    get_char ();
    set_border (menu_win, 0);
    test_menu ();

    delete_window (menu_win);
}

void main ()
{
    main_menu ();
}
```

Figure 4. Window Example

The example is written to facilitate later additions to the program, such as menus. The function main simply calls the function main_menu. The functions which_menu, test_menu, and main_menu each establish a window, set the border and color attributes of the window, display the window, write a list of menu items to the window, and delete the window. The function get_char (explained later) waits for the user to press a key and returns that key. Here it is used to pause the program.

Each function which_menu, test_menu, and main_menu demonstrates how the menus will operate. After each window is established, it is highlighted with a double line border until the user makes a choice (or, for now, presses a key). The window is then de-emphasized by changing it to a single line border, and the next function is called to demonstrate its window. When that function returns, the window is deleted and the function ends. The next step is to add menus to the example program to receive command input from the user.

Figure 5 shows the computer screen after all three windows are opened.

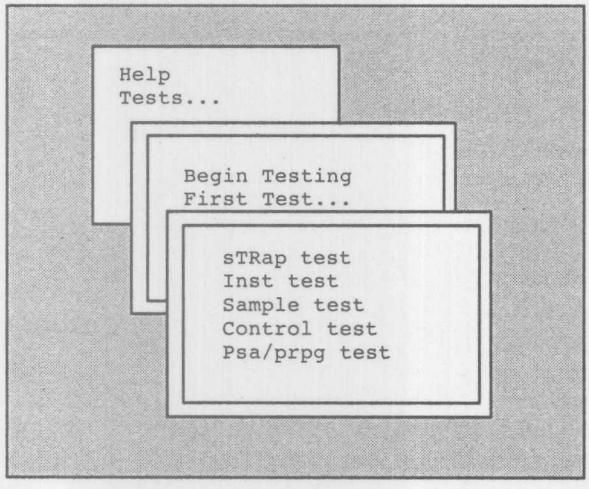


Figure 5. Screen with All Windows Opened

Using Menus

Menu Concepts

A menu is a list that contains the names of possible options the user can select. Once he selects one of those options, either a program function is executed or another menu appears from which additional options can be chosen. Menus are a common method for selecting program commands. They have the advantage of presenting all possible commands to the user up front, rather than requiring the user to remember the names of commands and type them in. Entering commands via menus often requires fewer keystrokes than typing in command names. And when users are unfamiliar with a program, menus reduce the number of times they need to reference the program documentation.

ASSET provides a simple method for implementing menus that is consistent with a windowing environment. A menu is a simply a window. The program writes the name of each menu item to the window and then calls a function to activate menu selection from the window. The user can choose a menu item by using the cursor keys to move a highlight bar up and down the menu until the highlight is over the necessary item and then press the Enter key to choose that item. Alternatively, the function allows the program to specify shortcut keys that, when pressed by the user, choose the same menu item without using the arrow keys or the Enter key. The user can also cancel the menu input by pressing the Escape key, in which case no menu item is selected.

Creating a Menu

Menus in ASSET leverage heavily off windows. To create a menu, the window functions are used to establish a window, set its attributes, and write the list of menu items, one per line, to the window. The version of the example program in Figure 1 does this already. Next, the function get_selection is used to activate menu processing. Its parameters include the window pointer, an integer defining the number of the menu item to place the highlight bar on, and a character pointer to a string containing the list of shortcut keys.

The parameters require further explanation. The initial menu item parameter lets the program determine which menu item should be highlighted initially. This can be used to always place the highlight bar on a specific menu item; to emphasize that item is the preferred choice. For example, if the menu was in response to a delete request, the "No - don't delete" menu item could always be highlighted for safety. The initial menu item parameter can also be used to remember which menu item was chosen last and to place the cursor back on that item. This can be used to speed up the selection of frequently-used menu items. The latter technique is used in the example program. The string of shortcut keys contains one character (preferably a letter or digit) for each menu item. If the user presses the first character when the menu appears, the first menu item is selected; the second character, the second item is selected; and so on. This feature can be disabled by passing NULL for the string pointer. When using shortcut keys, the shortcut key for each menu item should either be part of the name for the menu item or displayed next to it, and it should be emphasized in some way so the user realizes that it is a shortcut key.

Receiving Input From a Menu

Once the user makes a selection, the function get_selection returns an integer to indicate which menu item was chosen. If the user chose the first item, a 1 is returned; the second item, a 2 is returned; and so on. The number of the menu items is returned identically regardless of whether the user made the selection by using the arrow and Enter keys or by pressing the shortcut key. If the user decides not to make a selection by pressing the Escape key, a zero (0) is returned instead.

Once the selection is made, the program can delete the window to make the menu disappear.

Phase II of the Example: Menus

The source code in Figure 6 uses the get_selection function described earlier.

```
#include <stdio.h>
#include <stdlib.h>
#include <twindow.h>
#define FIRST_TEST 1          /* Test number of default first test */
#define LAST_TEST  5          /* Test number of default last test */
int first = FIRST_TEST;       /* First test procedure to be performed */
int last = LAST_TEST;         /* Last test procedure to be performed  */
void do_help (void)
{
    /* Your help facility would appear here */
    get_char ();
}
void do_tests (void)
{
    /* Your testing procedures appear here */
    get_char ();
}
    int which_menu (int selection)
{
    WINDOW *which_win;
    int new_selection;
    which_win = establish_window (5, 8, 7, 17);
    set_border (which_win, 1);
    set_colors (which_win, ALL, BLACK, WHITE, BRIGHT);
    set_colors (which_win, ACCENT, WHITE, BLACK, DIM);
    display_window (which_win);
    wprintf (which_win, " stRap test\n");
    wprintf (which_win, " Inst test\n");
    wprintf (which_win, " Sample test\n");
    wprintf (which_win, " Control test\n");
    wprintf (which_win, " Psa/prpg test");
    new_selection = get_selection (which_win, selection, "riscp");
    if (new_selection == 0)
        new_selection = selection;
    delete_window (which_win);
        return new_selection;
}
void test_menu (void)
{
    WINDOW *test_win;
    static int selection = 1;
    int new_selection;
    test_win = establish_window (3, 4, 5, 17);
    set_border (test_win, 1);
    set_colors (test_win, ALL, BLACK, WHITE, BRIGHT);
    set_colors (test_win, ACCENT, WHITE, BLACK, DIM);
    display_window (test_win);

    wprintf (test_win, " Begin testing\n");
    wprintf (test_win, " First test...\n");
    wprintf (test_win, " Last test...");

for (;;)
    {
        new_selection = get_selection (test_win,
            selection, "bfl");
        set_border (test_win, 0);
        if (new_selection == 1)       /*Begin testing menu item: */
            do_tests ();            /*Invoke the test procedure */
        else if (new_selection == 2)     /*First test menu item: */
            first = which_menu (first);  /*Choose which test*/
```

continued on next page . . .

```
        else if (new_selection == 3)      /* Last test menu item:*/
            last = which_menu (last); /* Choose which test*/
        else                              /* ESC key:*/
            break;              /*Leave the menu */

        selection = new_selection;
        set_border (test_win, 1);
    }
delete_window (test_win);
}
void main_menu (void)
{
    WINDOW *menu_win;
    static int selection = 1;
    int new_selection;
    menu_win = establish_window (1, 1, 6, 12);
    set_border (menu_win, 1);
    set_colors (menu_win, ALL, BLACK, WHITE, BRIGHT);
    set_colors (menu_win, ACCENT, WHITE, BLACK, DIM);
    display_window (menu_win);
    wprintf (menu_win, " Help\n");
    wprintf (menu_win, " Tests...\n");
    wprintf (menu_win, " Debug\n");
    wprintf (menu_win, " Exit");


    for (;;)
    {
        new_selection = get_selection (menu_win,
        selection, "htde");
        set_border (menu_win, 0);
        if (new_selection == 1)    /* Help menu item: */
            do_help ();          /*Invoke your help facility */
        else if (new_selection == 2)  /* Tests menu item: */
            test_menu ();          /*Display test menu */
        else if (new_selection == 3)  /* Debug menu item: */
            sc_debug ();           /*Invoke ASSET debugger */
        else                     /* Exit menu item and ESC key: */
            break;               /* Leave the menu  */
        selection = new_selection;
        set_border (menu_win, 1);
    }
    delete_window (menu_win);

}
main
{
main_menu ();
}
```

Figure 6. Menu Example

The main difference between this program and the version in Figure 1 is that previous calls to the function get_char are replaced by an infinite "for" loop that calls the function get_selection. The loop activates menu processing for each menu then de-emphasizes the menu by changing the window border to a single line. A cascading if statement performs different processing for each menu selection. The menu is again emphasized and the program loops back to receive more menu input. The final else of each if breaks out of the infinite loop, and the menu is closed by deleting the window and leaving the function.

The menus in the example program of Figure 3 "remember" the previous selection by using two variables for the menu selection. One of the variables, selection, is a static, local variable that contains the number of the previous menu item selected. Static variables retain their values between calls to the function, so the previous selection is always preserved. It is initialized to one (1) so that when the program is first run, the first menu item is highlighted when each menu is called. Each call to get_selection passes this variable as the initial menu item to highlight. The other variable, new_selection, is a local variable that contains the number returned by get_selection. It is assigned to selection after the menu selections and the Escape key are processed. Two variables must be used because when the Escape key is pressed, a zero (0) is returned. Zero is not a valid menu selection and will cause problems if it is passed to get_selection.

The menus in the example program also use shortcut keys as a convenience to the user. The letter of each menu item corresponding to the shortcut key is capitalized so that the user can quickly recognize the shortcut key. In all cases except one, the shortcut key is the same as the first letter of the menu item. This is the preferred choice, because the user will naturally assume it. The exception, "stRap test", was forced when two menu items began with the same first letter. Even in this case, the choice of "R" as the shortcut key is somewhat natural, because a strap is a Reset of the scan bus.

A few functions (do_help and do_tests) are added where additional program functions would appear. These functions are not relevant to the discussion here and are dummied out to avoid cluttering the text. In addition, the function sc_debug is called but does not appear in the source code. This function is part of the scan library supplied with ASSET and invokes the Run-Time debugger.

The figure below shows the computer screen with the test menu active and the highlight on the "First test" menu item.

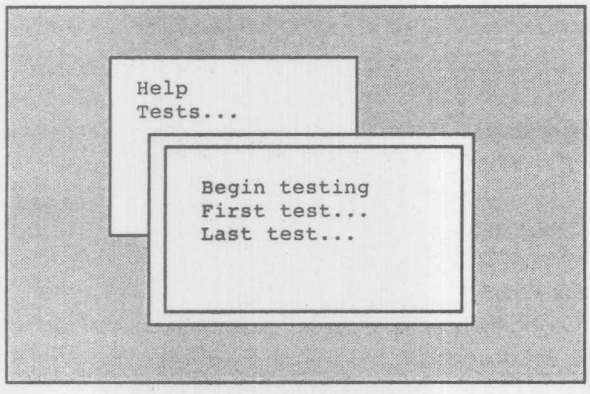


Figure 7. Screen with Menus Shown

Now all that remains of the example program is to prompt for the operator name.

Getting User Input

Data Entry Concepts

Programs need methods other than menus to receive input. This is especially true when all possible values for the input are not known. For example, it would be impossible for an employee database update program to present a list of all possible addresses in the world. For these situations, single characters or character strings need to be accepted.

Receiving User Input

ASSET provides two functions for receiving unstructured input: one for single characters and one for character strings. The function get_char is used to get keyboard input one key at a time. It waits for the user to press a key and then returns an integer that represents the value of that key. It accepts no parameters. The get_char is already used in the example program of Figure 1 to pause the program and wait for a key press.

For printable ASCII characters, the get_char function returns the ASCII code for that character. For nonprintable ASCII characters (such as control keys) or for key combinations that do not return ASCII values (such as the function or arrow keys), the function returns an integer beginning at 128. The file keys.h supplied with ASSET defines constants (like CTRL_A, F1, and UP) to represent these keys or key combinations.

The function get_char could be used to input strings but doing so would be inconvenient. The function getstring is used to input character strings. It establishes a window and displays a string buffer for editing. The user can edit the string buffer, using the arrow and editing keys, or type in a new string. When finished, the user presses the Enter key to accept the new string. Pressing the Escape key cancels the input operation, leaving the original string intact.

The getstring function parameters include a character pointer to a message string displayed as the window title, a character pointer to the string buffer to be edited, and an integer specifying the maximum size of the string buffer. It returns TRUE if the user presses Enter, or FALSE if Escape is pressed.

Phase III of the Example: Data Entry

The source code uses the getstring function to get the operator name for the program.

```
#include <stdio.h>
#include <stdlib.h>
#include <twindow.h>

#define FIRST_TEST 1       /* Test number of default first test */
#define LAST_TEST  5       /* Test number of default last test  */
#define NAME_LEN   80      /* Maximum length of operator's name */
int first = FIRST_TEST;    /* First test procedure to be performed*/
int last = LAST_TEST;      /* Last test procedure to be performed*/ char
op_name[NAME_LEN] = "";    /* Operatorname */

void do_help (void)
{
    /*Your help facility would appear here*/
    get_char ();
}
void do_tests (void)
{
    /* Your testing procedures appear here */
    get_char ();
}
int which_menu (int selection)
{
    WINDOW *which_win;
    int new_selection;

    which_win = establish_window (5, 8, 7, 17);
    set_border (which_win, 1);
    set_colors (which_win, ALL, BLACK, WHITE, BRIGHT);
    set_colors (which_win, ACCENT, WHITE, BLACK, DIM);
    display_window (which_win);
    wprintf (which_win, " stRap test\n");
    wprintf (which_win, " Inst test\n");
    wprintf (which_win, " Sample test\n");
    wprintf (which_win, " Control test\n");
    wprintf (which_win, " Psa/prpg test");

    new_selection = get_selection (which_win, selection, "riscp");
    if (new_selection == 0)
    new_selection = selection;

    delete_window (which_win);
    return new_selection;
}
void test_menu (void)
{
    WINDOW *test_win;
    static int selection = 1;
    int new_selection;
    test_win = establish_window (3, 4, 5, 17);
    set_border (test_win, 1);
    set_colors (test_win, ALL, BLACK, WHITE, BRIGHT);
    set_colors (test_win, ACCENT, WHITE, BLACK, DIM);
    display_window (test_win);
    wprintf (test_win, " Begin testing\n");
    wprintf (test_win, " First test...\n");
    wprintf (test_win, " Last test...");

    for (;;)
    {
        new_selection = get_selection (test_win, selection, "bfl");
        set_border (test_win, 0);
        if (new_selection == 1)        /*Begin testing menu item: */
            do_tests ();       /*Invoke the test procedure */
        else if (new_selection == 2)  /*First test menu item: */
            first = which_menu (first);    /*Choose which test */
        else if (new_selection == 3)      /* Last test menu item:*/
            last = which_menu (last); /*Choose which test */
        else                    /* ESC key:  */
            break;                  /*Leave the menu */
        selection = new_selection;
        set_border (test_win, 1);
    }

    delete_window (test_win);
```

continued on next page . . .

```
}
void main_menu (void)
{
    WINDOW *menu_win;
    static int selection = 1;
    int new_selection;

    menu_win = establish_window (1, 1, 6, 12);
    set_border (menu_win, 1);
    set_colors (menu_win, ALL, BLACK, WHITE, BRIGHT);
    set_colors (menu_win, ACCENT, WHITE, BLACK, DIM);
    display_window (menu_win);
    wprintf (menu_win, " Help\n");
    wprintf (menu_win, " Tests...\n");
    wprintf (menu_win, " Debug\n");
    wprintf (menu_win, " Exit");

    for (;;)
    {
        new_selection = get_selection (menu_win,
        selection, "htde"); set_border
        (menu_win, 0);
        if (new_selection == 1)    /* Help menu item:    */
            do_help ();          /*Invoke your help facility */
        else if (new_selection == 2)  /*Tests menu item: */
            test_menu ();          /*Display test menu  */
        else if (new_selection == 3)  /*Debug menu item: */
            sc_debug ();         /*Invoke ASSET debugger  */
        else                /* Exit menu item and ESC key: */
            break;             /* Leave the menu */
        selection = new_selection;
        set_border (menu_win, 1);
    }
    delete_window (menu_win);
}

main
{
    while (getstring ("Enter Operator Name", op_name, NAME_LEN))
    main_menu();
}
```

Figure 8. Data Entry Example

Only two areas of the program are changed for this example. At the top of the program, the constant NAME_LEN and the character array op_name are added to reserve space for the operator name. At the bottom of the program, in the main function, a loop is added to prompt for the operator name. If the user presses Enter after entering the operator name, the program brings up the main menu and loop back to the prompt afterwards. When the user is ready to leave the program, they must press Escape at the prompt, which will exit the loop and the program.

The figure below shows the computer screen with the prompt for the operator name filled in.

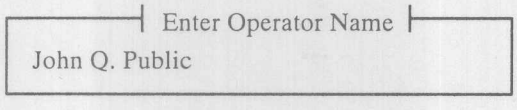


Figure 9. Operator Name

Conclusion

The user interface is a crucial aspect of program development. Working programs, even excellent programs, will be rejected if they cannot be operated easily. Designing a good user interface requires adherence to the principles of simplicity, consistency, and clarity. Simplicity makes the interface easier to understand, consistency makes it easier to remember, and clarity makes it easier to utilize. A good user interface increases the chances that a program will be accepted or even embraced by users.

ASSET provides a convenient library of functions that can be used to implement a commonly accepted style of user interface, the menu-and-window user interface. Combining these functions with the design principles mentioned earlier will result in programs that are easy to write and easy to use.

Design Tradeoffs When Implementing IEEE 1149.1

by Wayne Daniel

Introduction

This article explores the many design trade-offs when implementing the IEEE 1149.1 test bus in IC, Printed Wiring Boards (PWB), and systems. As with any design task, trade-offs must be made to determine the best mix of functionality, testability, reliability, producibility, and maintainability. As an industry-wide standard, the IEEE 1149.1 test bus and boundary scan architecture allows a consistent method for circuit controllability and observability across all levels of development and test. Test and debug operations can be performed via the IEEE 1149.1 test bus and boundary scan architecture in many areas including design verification, fault troubleshooting, factory board and system test, and field test without requiring actual physical access. Embedded test features, which previously could only be accessed at one test level, can easily be accessed at each level of integration (i.e., IC, board, subsystem, and system).

Test Considerations – Ad Hoc/Structured

There are basically two ways to apply testability to a design, Ad Hoc (or unstructured) and Structured. Ad Hoc testability is simply intended to solve a particular test problem (i.e., adding a test point to observe a signal line during board test). While this may be useful for a particular test environment, in this case factory board test, it may not be useful for any other test environment, such as IC or system test. Structured testability, on the other hand, is designed to be useful at several levels of test. For instance, an IC with Built-In Self-Test (BIST) and a standard test interface can be used for IC test, board test, and system test. This type of testability is considered structured because it is standardized and can be used in several test environments.

What Is IEEE 1149.1?

The IEEE 1149.1 test bus and boundary scan architecture allow an IC, and similarly a board or system, to be controlled via a standard four-wire interface. Each IEEE 1149.1 compliant IC incorporates a feature known as boundary scan that allows each functional pin of the IC to be controlled and observed via the four-wire interface. Test, debug, or initialization patterns can be loaded serially into the appropriate IC(s) via the IEEE 1149.1 test bus. This allows IC, board, or system functions to be observed or controlled without actual physical access.

The IEEE 1149.1 test bus is comprised of two main elements: a Test Access Port (TAP), which interfaces internal IC logic with the external world via a four-wire (optionally five-wire) bus; and a boundary scan architecture, which defines standard boundary cells to drive and receive data at the IC pins. The IEEE 1149.1 spec also defines both mandatory and optional opcodes and test features. The test bus signals are: Test Clock (TCK), Test Mode Select (TMS), Test Data In (TDI), Test Data Out (TDO), and the optional Test Logic Reset (TRST). The IEEE 1149.1 specification also specifies that the ICs can be connected in either a ring or star configuration. A simplified block diagram of the architecture is shown in Figure 1.

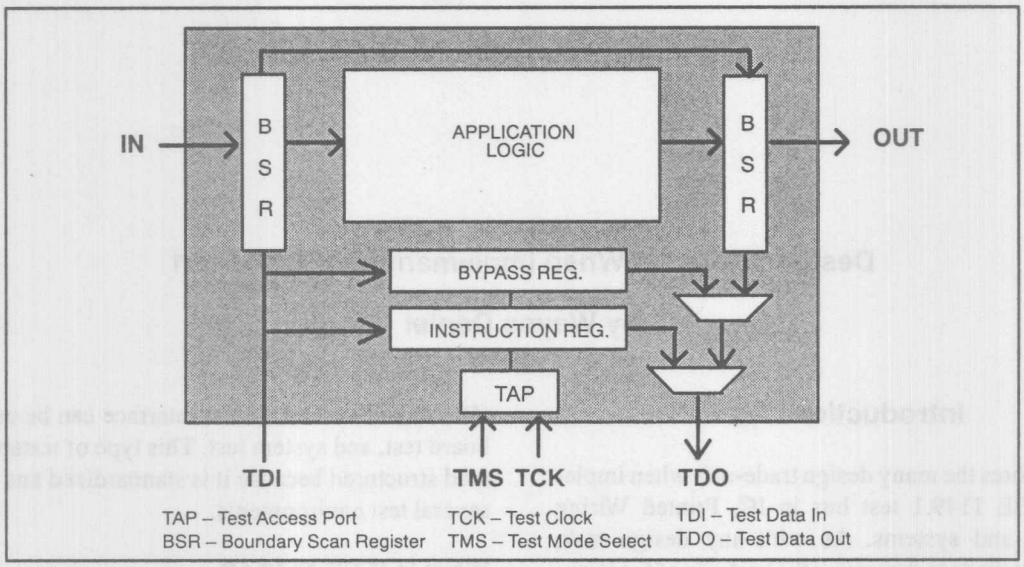


Figure 1. IEEE 1149.1 Scan Bus and Boundary Scan Architecture

Overview

Design Costs vs. Test Costs

Cost is a driving requirement when designing any system. Design costs are easily calculable in terms of part costs, manufacturing assembly, engineering design labor, etc. Tests costs are also quantifiable but may not be done as easily. Certainly test equipment and test software development costs can be readily calculated, but test times, troubleshooting, and repair are not as obvious. In complex designs or design that have long service lives, the Life Cycle Costs (LCC) of the product is dominated by test and maintenance cost, not design and production costs. It is typical to "invest" time and money to design for test which then saves production and maintenance costs. To provide an easy and cost effective method to include IEEE 1149.1 testability into any design, TI has developed a family of testable ICs and ASIC cells. These ICs and ASIC cells are members of the SCOPE (System Controllability, Observability, and Partitioning Environment) family.

Use of Scannable Parts

There are two complementary techniques when designing IEEE 1149.1 into a system: off-the-shelf parts (i.e, TI SCOPE octals), and Application-Specific Integrated Circuits (ASICs). Each satisfies a particular application, but both can be used together in the same design. TI SCOPE octals and other off-the-shelf parts are suitable when testability is needed in a functionally equivalent device. Consider a simple PWB with a processor and memory. Without testable SCOPE octals the address and data buses would have to be tested indirectly by writing and reading memory via the processor. The embedded address and data buses buffered by '244 and '245 parts can be replaced by functionally equivalent IEEE 1149.1 compliant TI SCOPE octals. Now, tests of the address and data buses can be achieved by controlling and observing the SCOPE octals via the IEEE 1149.1 test bus. Addresses and data can be explicitly loaded and driven by the SCOPE octals or functionally driven addresses and data can be captured and read via the IEEE 1149.1 test bus.

The other technique to incorporate IEEE 1149.1 features into a design is by designing it into ASICs. If ASICs are used, adding IEEE 1149.1 is a straightforward and efficient method of implementing testability in a design. To facilitate building IEEE 1149.1 compliant ASICs, TI provides ASIC library cells in the 1.0-μm standard cell and gate array libraries. TI SCOPE macros are available to implement the mandatory IEEE 1149.1 Test Access Port and boundary cells. Additionally, TI SCOPE cells are available that implement Pseudo-Random Pattern Generation (PRPG) and Parallel Signature Analysis (PSA) for use in IC BIST, board or system test.

TI also produces several other IEEE 1149.1 compliant test ICs to support PWB and system test. A Test Bus Controller (TBC) is available to drive the IEEE 1149.1 test bus. Other members of the SCOPE family include Scan Path Selectors (SPS) to partition long scan paths into multiple small chains, a Digital Bus Monitor (DBM) to monitor and capture data in real time, and ASSET hardware and software to control the test bus and simplify the task of developing scan-based test and debug software.

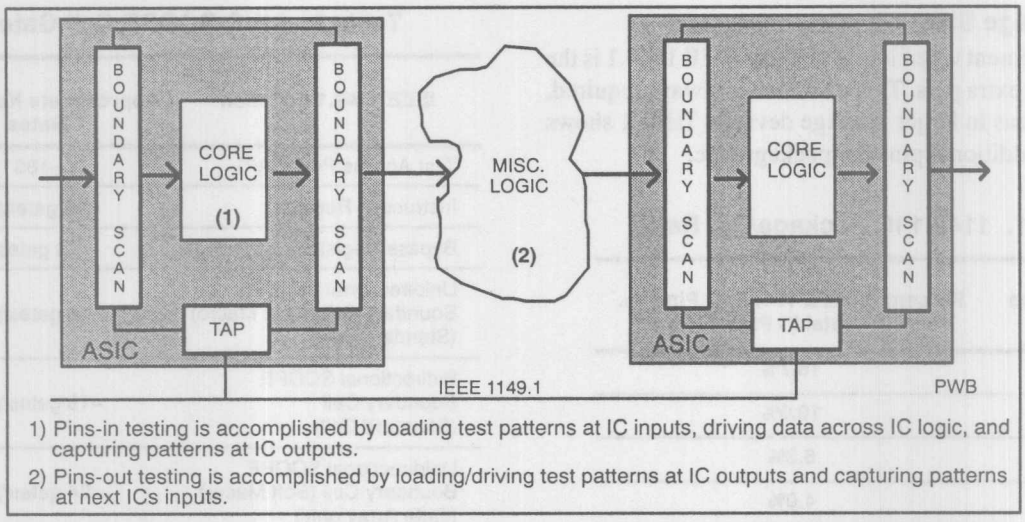


Figure 2. IEEE 1149.1 Pins-in and Pins-out Testing via Boundary Scan

Currently, TI has the following products available to implement IEEE 1149.1 in IC, board and system design and tests.

TI SCOPE cells — TSCS00, TGC100, TSC700
TI SCOPE Octals — SN74BCT8244, SN74BCT8245, SN74BCT8373, SN74BCT8374
IEEE 1149.1 Controllers — SN74ACT8990
IEEE 1149.1 Scan Path Selectors SN74ACT8997, SN74ACT8999
IEEE 1149.1 Digital Bus Monitor — SN74ACT8994
ASSET Hardware and Software

Design Tradeoffs in Implementing IEEE 1149.1 IC Level

The following paragraphs present IEEE 1149.1 design tradeoff information at the IC level. While this data is useful in simply comparing IEEE 1149.1 ICs to non-IEEE 1149.1 ICs, this one-to-one comparison is not as useful as comparing the total advantages/disadvantages at the PWB and system levels. Total PWB and system-level tradeoffs, not IC tradeoffs, should be used to determine the final implementation strategy.

Controllability and Observability

Obviously, in addition to performing their functional tasks, the SCOPE family of test products provide a level of controllability and observability that was previously unachievable. During design verification, test or troubleshooting, signal states may be sampled or controlled via the IEEE 1149.1 interface. Testing can be accomplished on the internal IC logic via pins-in testing, or tests on external IC logic or interconnects via pins-out testing. Pins-in testing is accomplished by loading IC input pins via the IEEE 1149.1 interface, driving the internal logic of the IC, capturing the output pin states and scanning out the results. Pins-in testing is accomplished without disturbing external IC pins. Pins-out testing is accomplished by loading IC output pins via the IEEE 1149.1 interface, driving the IC outputs and interconnects, capturing the data at the next IEEE 1149.1 ICs input pins and scanning out the results. Pins-in and pins-out testing is illustrated in Figure 2.

BIST/Additional Hooks

In addition to the standard four-wire interface and boundary scan architecture that IEEE 1149.1 defines, other test or debug "hooks" can be implemented and controlled. For ASICs it is very common to include some form of Built-In Self-Test (BIST) to assist in device testing. IEEE 1149.1 provides a standard interface to initiate BIST and retrieve results. Two common BIST methods known as Pseudo-Random Pattern Generation (PRPG) and Parallel Signature Analysis (PSA) are supported in the SCOPE test octal parts and ASIC standard cell libraries. PRPG provides the capability to generate pseudorandom patterns to stimulate a circuit under test. PSA allows a stream of patterns to be collected and compressed into a unique signature. In addition to boundary scan, ASICs and other special purpose ICs may incorporate internal scan within the IEEE 1149.1 architecture. Internal scan may be used to partition the IC into more easily testable functions, write/read internal registers, etc. TI's new digital signal processors, the TMS320C50, incorporates internal scan to provide built-in in-circuit emulation features for test and debug. Via the 'C50's IEEE 1149.1 interface, registers can be loaded/examined and the processor execution can be controlled to RUN/STOP, SINGLE-STEP, etc.

IC Pins/Package Size

The first requirement when implementing IEEE 1149.1 is the addition of four extra pins. This overhead is always required, but is less obvious in larger package devices. Table 1 shows the percent of additional pins per package size.

Table 1. 1149.1 IC Package/Pin Ratio

IC Package Size	Percent of IEEE 1149.1 IC Pins vs. Total IC Pins
24 pins	16.7%
40 pins	10.0%
64 pins	6.3%
100 pins	4.0%
132 pins	3.0%
160 pins	2.5%
208 pins	1.9%

Gate Count

The number of gates to implement IEEE 1149.1 is primarily driven by the number of IC I/O pins. The reason is because IEEE 1149.1 requires each functional I/O pin to have a boundary cell. Naturally, low gate count ICs with a high number of I/O pins will have proportionally more gates to implement IEEE 1149.1. Some typical gate counts from the 1.0-μm standard cell and gate array library are shown in Table 2. The last SCOPE I/O boundary cell in the table is a soft macro that is typically used in gate arrays. The IEEE 1149.1 specification requires a 2-bit Instruction Register (IR) as a minimum, but longer IRs are allowed to implement additional application-specific instructions. In addition to the basic SCOPE cells indicated above, the TI SCOPE octals and SCOPE ASIC library also provides additional features such as PRPG and PSA incorporated in the SCOPE cell.

Table 2. ASIC SCOPE Cell Gate Count

IEEE 1149.1 Function	Approximate Number of Gates
Test Access Port (TAP)	~183
Instruction Register	~20 gates/bit
Bypass Register	~8 gates
Unidirectional SCOPE Boundary Cell (Hard Macro) (Standard Cell)	~15 gates/pin
Bidirectional SCOPE Boundary Cell (Standard Cell)	~19 gates/pin
Unidirectional SCOPE Boundary Cell (Soft Macro) (Gate Array only)	~24 gates/pin

The formula below can be used to estimate the IEEE 1149.1 gate count overhead. (Remember to only count the number of functional I/O pins, not power, ground, and unused pins).

Total IEEE 1149.1 Gate count =
*(TAP) + [(IR) * (IR bit width)]*
*+ (BR) + [(# I/O pins) * (# gates/pin)]*

Figures 3 and 4 show the relationship between functional gate count vs. IEEE 1149.1 test logic vs. pin counts increase for both standard-cell and gate-array designs.

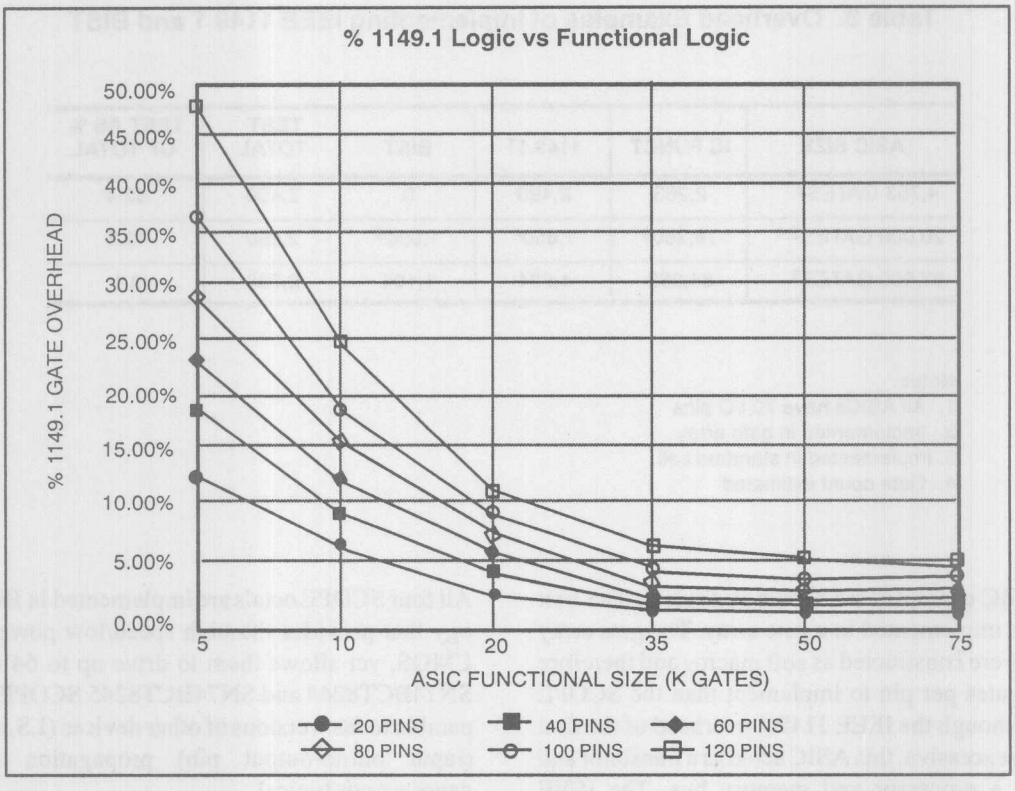


Figure 3. Standard Cell ASIC 1149.1 Overhead

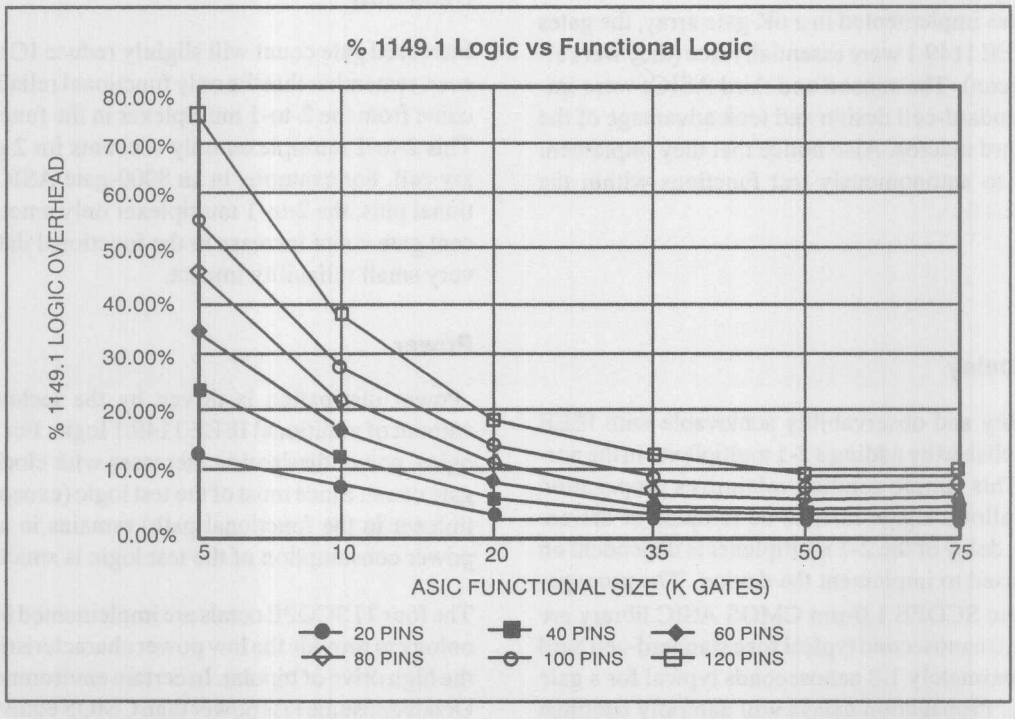


Figure 4. Gate Array ASIC 1149.1 Overhead

Table 3. Overhead Examples of Implementing IEEE 1149.1 and BIST

ASIC SIZE	IC FUNCT	1149.1[1]	BIST	TEST TOTAL	TEST AS % OF TOTAL
4,753 GATES[2]	2,263	2,490	0	2,490	52.4
20,000 GATES[3,4]	18,350[4]	1,650[4]	1,000[4]	2,650	13.3
87,000 GATES[3]	84,262	1,634	1,104	2,738	3.1

Notes:
1. All ASICs have 70 I/O pins
2. Implemented in gate array
3. Implemented in standard cell
4. Gate count estimated

Some actual ASIC examples are shown in Table 3. The first small ASIC was implemented in a gate array. The gate array boundary cells were constructed as soft macros and therefore required more gates per pin to implement than the SCOPE hard macros. Although the IEEE 1149.1 overhead of the first ASIC may seem excessive, this ASIC acted as a translator and buffer between a processor and memory bus. The IEEE 1149.1 boundary scan provided partitioning and allowed independent testing of the the functions via IEEE 1149.1. Since the first ASIC was implemented in a 6K gate array, the gates to implement IEEE 1149.1 were essentially free (they were already on the silicon). The second and third ASICs were implemented as standard-cell design and took advantage of the boundary-cell hard macros. Also notice that they implement BIST functions to autonomously test functions within the ASICs.

Propagation Delay

The controllability and observability achievable with IEEE 1149.1 is accomplished by adding a 2-1 multiplexer in the normal data path. This simple solution minimizes propagation delays, yet still allows signal lines to be sampled or driven. The propagation delay of the 2-1 multiplexer is dependent on the technology used to implement the device. The propagation delays for the SCOPE 1.0-μm CMOS ASIC library are approximately 1.0 nanosecond typical for a standard-cell hard macro and approximately 1.8 nanoseconds typical for a gate array soft macro. Propagation delays will naturally continue to decrease as technology matures.

All four SCOPE octals are implemented in BiCMOS technology that provides the high speed/low power advantages of CMOS, yet allows them to drive up to 64 milliamps. The SN74BCT8244 and SN74BCT8245 SCOPE octals are comparable to fast versions of other devices (LS, AS) with A-to-B (input pin-to-output pin) propagation delays of 6.0 nanoseconds typical.

Reliability

Increased gate count will slightly reduce IC reliability. However, remember that the only functional reliability impact will come from the 2-to-1 multiplexer in the functional data path. This 2-to-1 multiplexer only accounts for 2 gates per boundary cell. For example, in an 8000-gate ASIC with 100 functional pins, the 2-to-1 multiplexer only amounts to a 2.5 percent gate count increase in the functional data path. This is a very small reliability impact.

Power

Power dissipation is driven by the technology used and amount of additional IEEE 1149.1 logic. For CMOS technologies, power dissipation increases with clock frequency and gate count. Since most of the test logic (except the 2-to-1 multiplexer in the functional path) remains in a static state, the power consumption of the test logic is small.

The four TI SCOPE octals are implemented in BiCMOS technology to provide the low power characteristics of CMOS and the high drive of bipolar. In certain environments, the SCOPE Octals consume less power than CMOS equivalents, as shown in Figure 5.

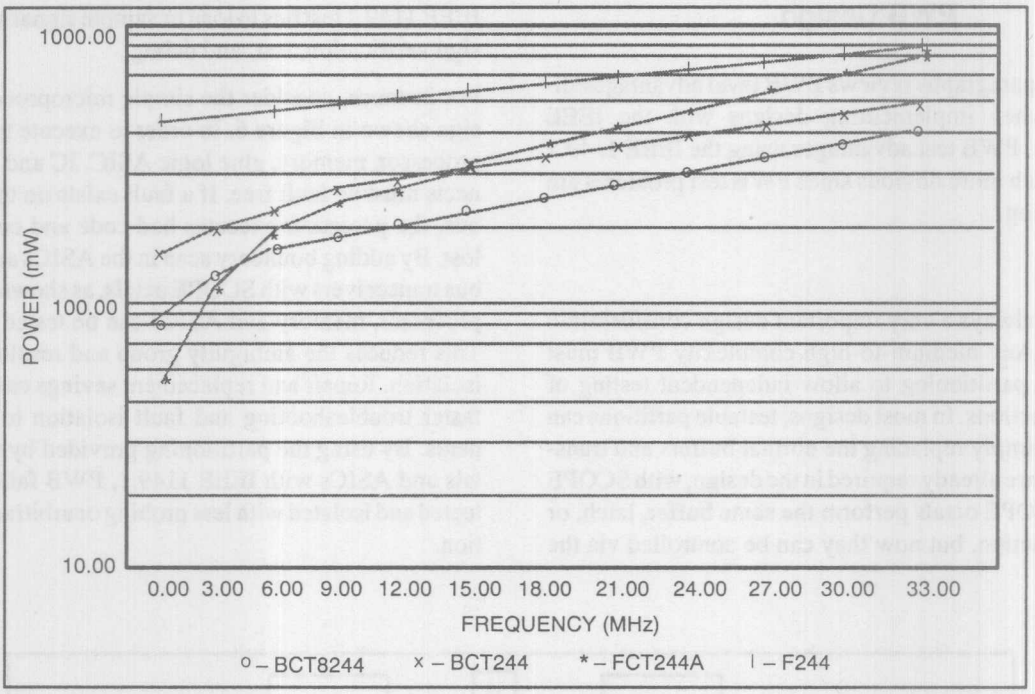


Figure 5. SCOPE BCT8244 Octal Power Consumption

Test Costs

The costs of IC test for IC production test and incoming inspection varys greatly with the complexity of the IC. For simple ICs such as the SCOPE octals, there is only slight advantages in using IEEE 1149.1 in IC level test. For medium to high complexity ICs such as ASICs and VLSI devices, IC test using IEEE 1149.1 provides several benefits. First, ASIC and VLSI devices can be tested statically or at low frequencies using the IEEE 1149.1 pins-in and pins-out test methods. Actually, an IEEE 1149.1 based pins-in tester has been demonstrated that simply consists of an IC socket with power, ground, and an IEEE 1149.1 test bus controller. Tests patterns are loaded and captured inside the IC via the IEEE 1149.1 test bus. This method provides a quick, inexpensive method to verify basic IC functionality.

IC test development costs can be greatly reduced by using the same test patterns used for IC design verification, simulation, and IC tests. The simulation patterns or a subset are applied via the IEEE 1149.1 test bus as previously described. The second major benefit of the IEEE 1149.1 test bus is the ability to control and examine internal nodes or states of ASIC and VLSI devices. Hard to test functions can be partitioned via internal scan and tested independently with smaller, easier to test partitions. An ASIC that may require 2^{20} (1 million) test patterns just to test a 20-bit counter can be tested in a fraction of the patterns by implementing internal scan on the counter. Patterns can be loaded directly via scan to test any count sequence. A recently developed ASIC with very long counter chains saved over four million test patterns by implementing internal partitioning controlled via the IEEE 1149.1 test bus.

IC Costs

IC purchase costs with IEEE 1149.1 are higher than their equivalent untestable versions. The cost delta varys depending on the proportion of IC pins and test logic to functional pins and logic. For larger ASICs the cost increase is small, but even in small ASIC designs the PWB and system test benefits are realizable. For I/O limited ASICs (ASICs that have unused core gates), the cost increase is typically less than the proportional gate increase. For core limited ASICs (ASICs that do not have spare gates), the cost may be proportional to the increase in gates or possibly higher.

In the case of the SCOPE octals, the cost is approximately three times the cost of the conventional IC. It must be remembered that in addition to performing the functional task, the SCOPE octals also add additional test capabilities. Costs will vary depending on the specifics of the design and market conditions. Check with your TI representative for details.

PWB Design

The following paragraphs reviews PWB level advantages/disadvantages when implementing designs with the IEEE 1149.1 test bus. PWB test advantages using the IEEE 1149.1 test bus are much more obvious since PWB test problems are more challenging.

Partitioning

Partitioning is always a very important design consideration for PWB test. Any medium-to-high complexity PWB must have adequate partitioning to allow independent testing of major logic functions. In most designs, testable partitions can be created by simply replacing the normal buffers and transceivers, which are already required in the design, with SCOPE octals. The SCOPE octals perform the same buffer, latch, or transceiver function, but now they can be controlled via the IEEE 1149.1 test bus to load or sample signal states during design verification, test, and debug.

For instance, consider the simple microprocessor board design shown in Figure 6. In order to execute test software the processor, memory, glue logic ASIC, IC and PWB interconnects must be fault free. If a fault exists on the memory data bus, the processor executes bad code and control would be lost. By adding boundary scan in the ASICs and replacing the bus transceivers with SCOPE octals, as shown in Figure 7, the processor, memory and ASICs can be tested independently. This reduces the ambiguity group and results in better fault isolation. Repair and replacement savings can be realized by faster troubleshooting and fault isolation to fewer components. By using the partitioning provided by the SCOPE octals and ASICs with IEEE 1149.1, PWB failures can be detected and isolated with less probing or arbitrary part substitution.

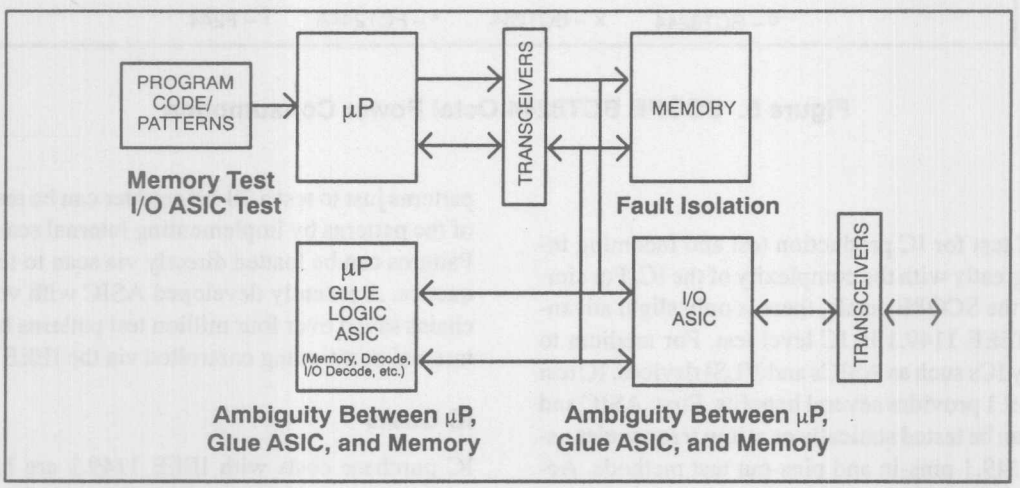


Figure 6. Simple Processor PWB Design

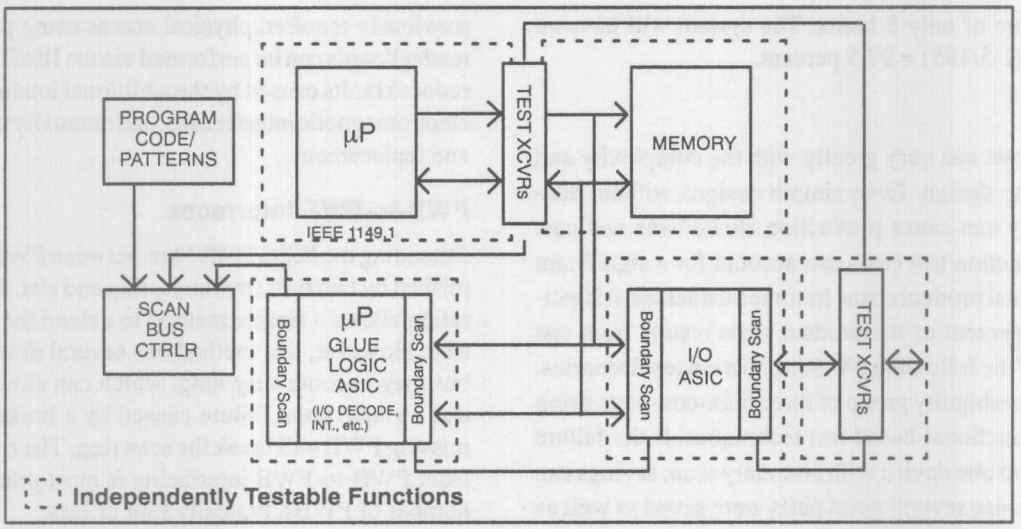


Figure 7. Design Partition via Boundary Scan

Real Estate

The PWB test logic real estate impact can be minimized, and, in some cases, PWB real estate can be gained by using IEEE 1149.1 to test functions. PWB real estate savings can be accomplished by replacing an IC added for test purposes with boundary scan embedded in the IC silicon. Test logic that is added to meet fault detection or fault isolation can be efficiently implemented and controlled via the IEEE 1149.1 test bus.

In a recent design, several latches were added to capture and buffer some key internal bus control signals for PWB test. If the internal bus is buffered by ASICs with boundary scan or SCOPE octals, the signal states can be observed via the boundary scan cells. This eliminates the need to add components for test purposes.

Test Points/Connector Size

IEEE 1149.1 boundary scan can be used to reduce or eliminate the number of test points or test pins on a PWB. When the four-wire IEEE 1149.1 test bus is brought out to the connector many previously "hidden" internal nodes become visible. Boundary scan cells can be thought of as virtual test points that can sample or control a node as shown in Figure 8. These virtual test points allow signal states to be scanned out and examined. Similarly, signal states can be scanned in and driven across circuit logic and interconnects. These control and observe operations can be performed via the IEEE 1149.1 test bus without the need to physically probe or route the signals under test to a test connector.

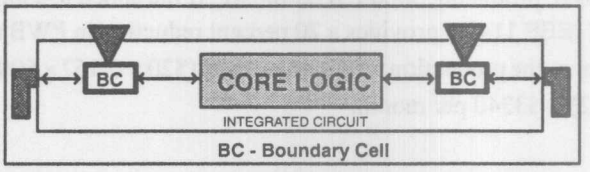


Figure 8. Virtual Test Points via IEEE 1149.1 Boundary Scan

Two main advantages are obvious when using IEEE 1149.1 boundary scan as virtual test points. The first advantage is that less test points are required if critical signals are buffered by SCOPE octals or ASIC boundary cells. The second advantage is that boundary cells are not subject to the potential noise problems that may be caused by additional etch and pins of test points.

Reliability

PWB reliability usually will not increase significantly by adding IEEE 1149.1 to the design. The small increase in silicon gates may not be notable when compared at the PWB level. In fact, reliability may increase when Ad Hoc testability is replaced by IEEE 1149.1. Also, remember that the only impact-to-device functional logic reliability is the addition of the 2-to-1 multiplexer in the data path.

Another key factor to consider along with reliability is system availability. A small decrease in system reliability may not be important if system availability increases. Consider a system with a 200-hour MTBF and a repair time of 10 hours. The system will have an availability of (1-10/200) = 95 percent. Now consider a more testable system that has a 195-hour MTBF

and a repair time of only 5 hours. The system will have an availability of (1-5/195) = 97.5 percent.

Test Costs

PWB test costs can vary greatly with the complexity and testability of the design. Even simple designs without adequate testability can cause production slowdowns and cost overruns. Production test costs can account for a significant portion of the final product costs. In some businesses, it is estimated that 25 percent of the product costs results from test costs. Consider the following PWB production test scenarios. A PWB has an ambiguity group of three high-cost parts using conventional functional-based test techniques. If the failure can be isolated to one device with boundary scan, savings can be realized because several good parts were saved as well as the labor and time required for unnecessary replacement. In another scenario, an hour of technician's labor including overhead is $20/hour, and a PWB test requires 10 minutes per PWB, producing 5000 PWBs/month. If the added testability of IEEE 1149.1 provides a 20 percent reduction in PWB test times, the production test savings equal ($20 × 0.167 × 5000 × 0.2) = $3340 per month.

System Test

IEEE 1149.1 can also be used for system level tests. The increased access afforded by the test bus and boundary scan allow control and observability in a "closed" system. Tests that previously required physical access using probe clips or extender boards can be performed via the IEEE 1149.1 bus. This reduces faults caused by the additional loading of test probes, electromagnetic interference, and manually inducted removal and replacement.

PWB-to-PWB Interfaces

Extending the IEEE 1149.1 bus between PWBs can be accomplished by two basic methods, ring and star. The ring configuration allows a simple method to extend the PWB level scan ring. However, this method has several drawbacks; the scan bus may become very long, which can slow test throughput and a single point failure caused by a broken connection or missing PWB will break the scan ring. The ring configuration from PWB-to-PWB interfacing is most practical for a small number of PWBs, typically four or less.

An IEEE 1149.1 star configuration between PWBs allows each PWB scan ring to be addressed without the overhead of additional PWBs in the scan path. However, this method requires additional backplane signals and only allows one PWB at a time to be addressed. Multiple PWBs cannot be scanned simultaneously with a star configuration.

There are several methods to efficiently partition scan rings from PWB-to-PWB. The simplest solution is to use the ACT8994 or ACT8997 scan path selectors to partition PWB scan rings. These devices allow a complete PWB scan ring to act as one device in bypass mode. This shortens a PWB scan ring to just 1 bit in the scan path. For more information on using scan path selectors see the article "Partitioning Designs with 1149.1 Scan Capabilities."

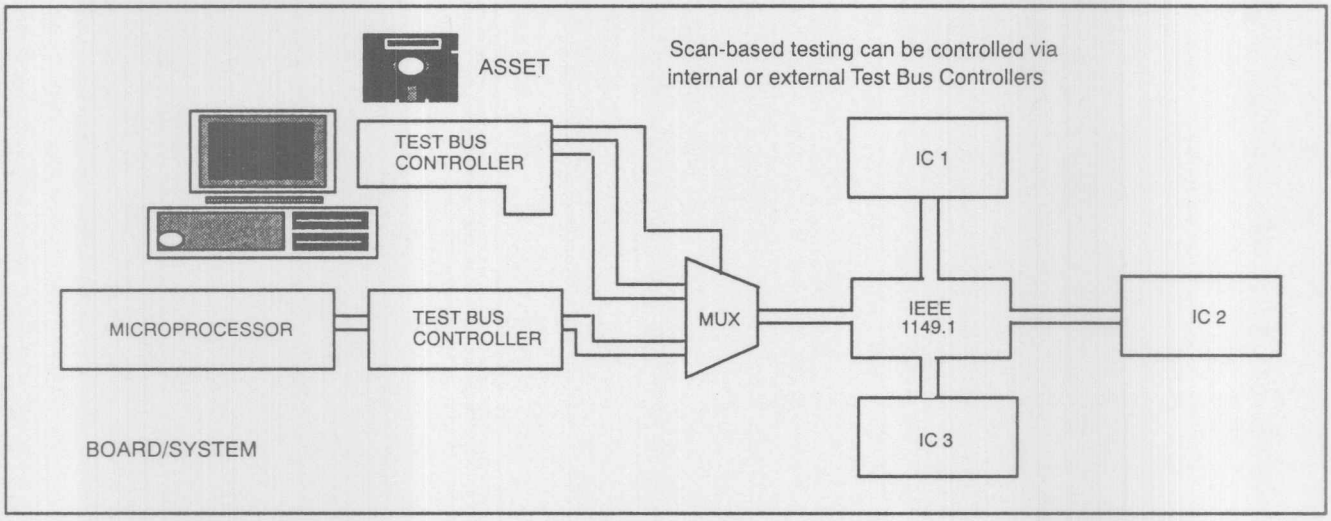


Figure 9. IEEE 1149.1 Test Bus Controllers

Test Bus Controllers

Control of the IEEE 1149.1 test bus for system test can be accomplished via either an external test bus controller (TBC), as in TI's PC-AT based ASSET system, or via an internal embedded test bus controller. Figure 9 shows an example system with internal and external TBCs. For factory- or maintenance-type testing, a single external TBC will suffice as the IEEE 1149.1 master. Naturally, in this configuration scan based tests can only be performed under external control.

The other option, an embedded TBC, allows autonomous testing under control of the embedded TBC. For small systems (less than four boards), a single TBC may be sufficient to test the system quickly. For larger or more complex systems, multiple TBCs may be required to test the system within an allocated time limit. The actual implementation method depends on the requirements for test execution time, real estate limits, and fault tolerance.

Conclusions

Considering all the advantages that a standard test bus and boundary scan architecture provides, IEEE 1149.1 should be seriously considered as a test solution. While the capabilities gained are not free, the tradeoffs should be investigated. Advantages include increased controllability and observability, test reuse, better fault detection and isolation, and consistent test methods across multiple test environments. Impacts include cost, propagation delay of one 2-1 multiplexer in the signal path and increased gate count for test logic. Although IEEE 1149.1 is suitable for all logic design sizes, implementing IEEE 1149.1 is typically easier to justify on larger designs. In the tradeoff analysis consider "hidden" costs, such as fault isolation size, test development time, test execution time, repair times, and life cycle repair and maintenance costs. These costs should not be underestimated. Using the capabilities of the IEEE 1149.1 test bus and boundary scan will provide advantages that help to reduce the total cost of ownership.

Hardware and Software Integration and Debugging Using ASSET

by Daniel R. Fusting

Introduction of Boundary Scan/Control

With the increasing complexity and density of electronic circuits, assembly procedures, and ASIC implementations, access to every part of a circuit for testing is becoming increasingly difficult. Alternate methods of accessing nodes within a design are necessary as circuit geometries continue to shrink while the demand for maintenance and testing becomes greater. One way to facilitate designed-in testing is a serial scan test bus, which allows access to all levels of a design, from ASIC internal nodes to a circuit board or system level. In addition to the benefits derived from integrating test scan into a circuit, designers should consider other significant benefits:

- Better circuit partitioning can be realized, leading to a hierarchical, structured debugging methodology.
- A "hands-off" approach, as nodes can be observed and controlled via a remote host with easily-developed test routines.
- Tests developed for various sections of the circuit can be used in subsequent phases of the design cycle, whether going from ASIC tests to board-level tests or from debugging to production tests.
- Design, fabrication, and assembly problems such as opens, shorts to power or ground, and signals shorted together can be easily located using scan techniques.
- Test functions that may have been temporarily wired onto a board (resets, clocks, etc.) can be controlled via software and the scan bus.
- Functional emulation of circuit interfaces can be accomplished through scan operations, allowing off-line testing and design development regardless of missing or nonfunctional external circuitry.

Throughout the process of debugging a circuit, serial scan paths provide a natural "divide and conquer" approach to design verification. In fact, increased accessibility through scan circuitry allows a greater degree of fault isolation while simultaneously reducing the dependance on logic analyzers, oscilloscopes, and the problems resulting from introducing these test components into the circuit. Test functions that had to be temporarily wired onto a board (resets, clocks, etc.) using earlier test methods can now be controlled via software and the scan bus.

1149.1 Overview

The IEEE has approved a standard for a four-wire serial scan bus for built-in testing of designs. A key feature of this bus, 1149.1 (Joint Test Action Group), is the concept of boundary scan. Boundary scan technology includes scan cells at the pins of a device so that the pins can have a "snapshot" taken of their state without interrupting their normal circuit functions. The pins can also be controlled via software and one of several 1149.1 test modes. Boundary scan allows for the control and observation of all the pins of a device with designed-in 1149.1, while preserving the fully functional state of the device when it is inactive or not in a test mode. When fully integrated into a design, boundary scan provides an unprecedented level of nonintrusive testability.

SCOPE Overview

Texas Instruments endorses 1149.1 as the solution for integrated test through a serial scan bus, and has a wide range of development tools to support designs with integrated 1149.1 capability. Collectively, these hardware and software tools comprise the SCOPE (System Controlability, Observability, and Partitioning Environment) environment. For ASIC applications, TI offers macros in their standard cell library to implement I/O cells with boundary scan, and a Test Access Port controller (TAP). For board level applications, TI offers several components that have 1149.1 capability, including the SCOPE octals (BCT8244, BCT8245, BCT8373, and BCT8374) and other chips that support a hierarchical implementation of 1149.1 (such as the Scan Path Selector, which controls switching between multiple parallel scan paths). To interface to a system or design with 1149.1 capability, Texas Instruments offers a Test Bus Controller chip (74ACT8990),

which provides a generic processor interface to implement an 1149.1 scan bus master function. To control the 1149.1 test system, TI provides the ASSET integrated software environment. Operating within a standard C++ development toolset, ASSET provides a level of abstraction and a library of utilities that ease 1149.1 test environment requirements.

The Importance of Considering Test/Scan Throughout the Design Phase.

At the beginning of the design cycle, consideration must be given to the architecture of the scan path(s). Decisions such as what functional blocks of logic should be surrounded by 1149.1 ICs, the number of parallel scan paths necessary, which circuits should be included on each ring or scan path, and what signals need to be controlled by 1149.1, have to be made in order to optimize scan bus operations. The design of the scan bus architecture should be a structured, parallel effort that complements the natural top-down design of the functional circuitry. Close cooperation between the design and testability engineers is essential during these phases of the project.

The Functional Circuitry/Test Bus Relationship

A basic philosophy of the 1149.1 standard is the preservation of functional circuit operation while the test bus is not in an active test mode. When first powered up, an 1149.1 test bus always comes up in an idle state, allowing the system to function normally. During normal circuit operation, any number of scan operations can be performed to load test parameters and commands into a component's scan path or sample data, without interference to the functional circuit. It is important to note that not all devices on a scan path must be in a test mode simultaneously. For instance, it is possible to isolate and test a single ASIC on a board, or a single board in a system, while the remaining circuitry continues to function normally. It is also possible to control a portion of a circuit and use it to test other functional blocks of the design using ASSET routines, or to initiate and retrieve results from BIT (Built-In-Testing).

Using ASSET to Control Circuit Behavior

By introducing ASSET and 1149.1 into a design, various modes of test and debug that were previously not possible become available through a software-only method. These include individual control of nodes within a logic block, static-level control of signals for manual probing, Parallel Signature Analysis and Pseudo-Random Pattern Generation (PSA/PRPG), and automated test routines for periodic BIT and production environment testing.

```
void mfm::reset_MFM (void)

//  Name:     mfm::reset_MFM      (void)
//  Description:
//  Scan a zero and then a one to the VME reset signal to generate an
//  active low reset pulse. vme_rst_i is defined to be U78.output(7,7).

{
u9.select_scan (VME_BUD);      // select VME scan path
u78.extest ();                 // select boundary scan operation
sc_scan (IR);                  // scan in extest opcode
u78.enable ();                 // be sure the octal gets turned on
vme_rst_i = 0;                 // pull u78 VRST~ output low
sc_scan (DR);                  // scan data out
vme_rst_i = 1;                 // drive u78 VRST~ output high
sc_scan (DR);                  // scan data out
u78.bypass ();                 // return to normal mode
sc_scan ( IR );                // scan bypass out
}
```

Figure 1. Software Reset Performed by Scan Operations

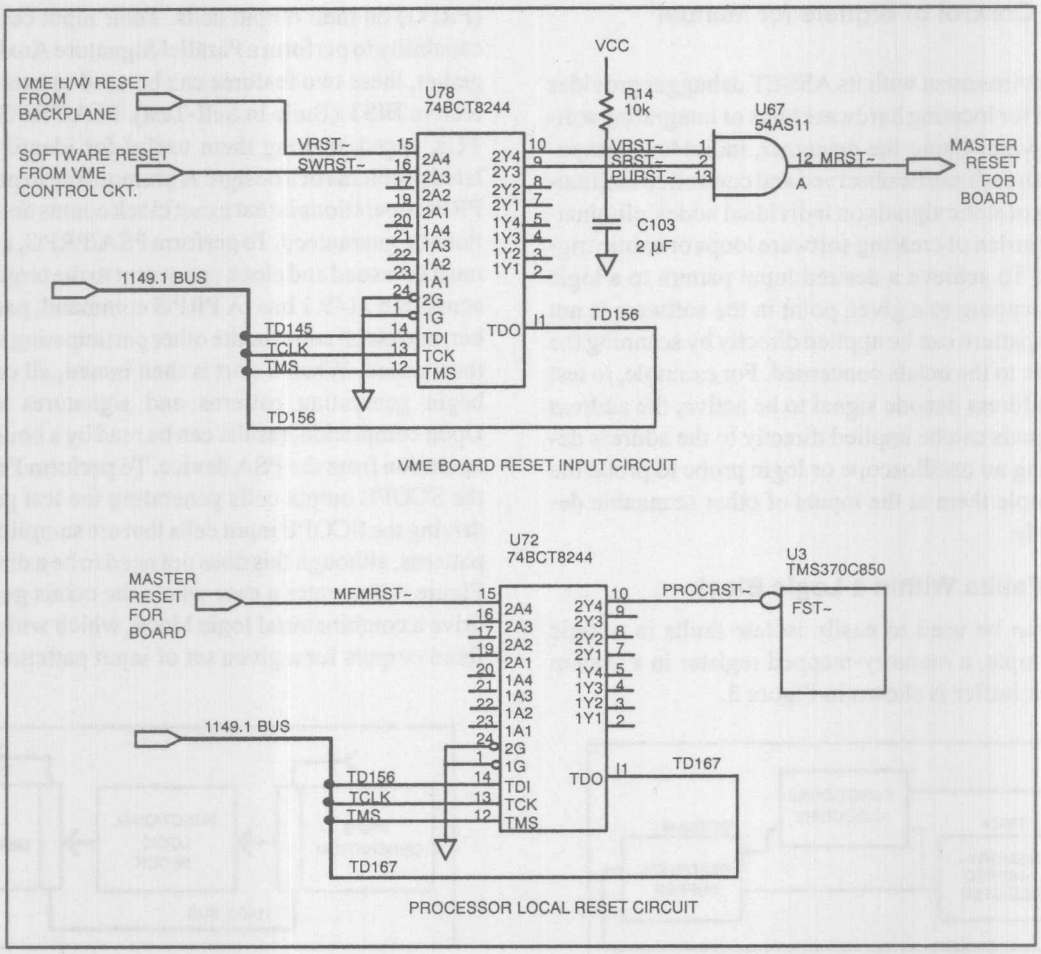


Figure 2. VME Board and Processor Reset Circuits

Control of Individual Nodes

One of the fundamental principles of 1149.1 is direct control of output pins of a device through boundary scan. This can result in simple yet powerful debugging and integration functions when applied to key signals. As an example, consider reset functions. It is often desirable throughout the debugging, integration, and software development phases of a design to have the capability to return to a reset state. This is easily implemented in the ASSET environment by performing boundary scans to a target device that buffers a reset signal. The software required to perform this procedure is shown in Figure 1, and the corresponding circuit is shown in Figure 2.

These scan operations select the boundary registers, scan a logic 0 to the reset output, scan a logic 1 to the reset output, and then, return the target device to its functional state.

Note the advantages to this approach:

- No external hardware such as reset switches or de-bounce circuits has to be attached to perform this operation.
- The resulting reset pulse is inherently de-bounced and clean.

More than one reset may exist, affecting different partitions of a system, or the system as a whole. As an example, consider a board with an embedded processor that plugs into a VME backplane. Three reset sources that exist in this scenario are shown in Figure 2. The VME backplane hardware reset signal (vrst~), software reset (swrst~), and power-up reset (purst~) can each trigger a total board reset (mfmrst~). By buffering the board reset to the processor with a SCOPE BCT8244, as shown in Figure 2, it is possible to scan a reset only to the processor without affecting the rest of the board reset circuitry.

Static-Level Control of Signals for Manual Probing

The SCOPE environment with its ASSET debugger provides a powerful tool for locating hardware bugs or integrating software and hardware. Using the debugger, individual components on the scan path can be observed and controlled facilitating the probing of static signals on individual nodes, eliminating the added burden of creating software loops or exotic trigger conditions. To achieve a desired input pattern to a logic block, single stepping to a given point in the software is not necessary. The pattern can be applied directly by scanning the appropriate data to the octals concerned. For example, to test for a specific address decode signal to be active, the address and control signals can be applied directly to the address decode logic, using an oscilloscope or logic probe to probe the outputs, or sample them at the inputs of other scannable devices if available.

Isolation of Faults Within a Logic Block

Scan control can be used to easily isolate faults in a logic block. For example, a memory-mapped register in a system with a readback buffer is shown in Figure 3.

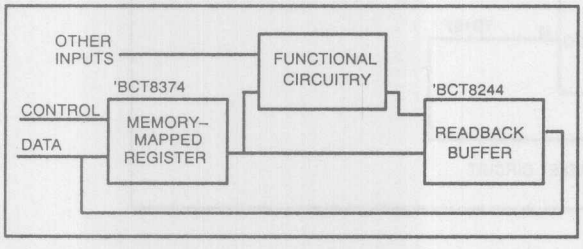


Figure 3. Memory-Mapped Register With Readback and Scan

Several BIT options become available by making the register and the readback buffer scannable components. If a write to the register cannot be verified through a subsequent read operation, a scan test mode can be entered to isolate the problem. If the register is a scannable component, it can be sampled to determine if the data was written successfully, isolating any problems in the paths up to the register. If the register contents can be verified, its outputs can then be verified by sampling the data inputs on the readback buffer. This operation verifies the integrity of any functional logic between the register and the actual readback data buffer. If this data is also found to be valid, scan testing can be used to place data patterns on the output of the readback buffer to test the paths back to the host.

Performing PSA/PRPG Operations

Texas Instruments' SCOPE products are designed with the capability to perform Pseudo-Random Pattern Generation (PRPG) on their output cells. Their input cells also have the capability to perform a Parallel Signature Analysis (PSA). Together, these two features can be used to execute fast and effective BIST (Built-In Self-Test). PSA/PRPG tests are run at TCK speed, making them useful for identifying timing-related problems of a design. A prerequisite to performing PSA/PRPG operations is that exact clock counts and starting conditions be guaranteed. To perform PSA/PRPG, a PSA command must be issued and clock count sent to the targeted component across the 1149.1 bus. A PRPG command, pattern seed number, and clock count to the other participating components are then issued. When a start is then issued, all components will begin generating patterns and signatures simultaneously. Upon completion, results can be read by a boundary scan read operation from the PSA device. To perform PSA/PRPG tests, the SCOPE output cells generating the test patterns must be driving the SCOPE input cells that are sampling the incoming patterns, although this does not need to be a direct connection. Figure 4 illustrates a case where the octals generating PRPG drive a combinatorial logic block, which will provide known fixed outputs for a given set of input patterns.

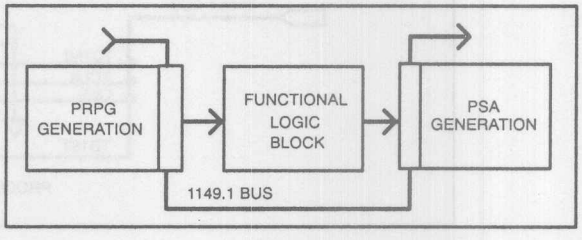


Figure 4. PSA/PRPG Generation Through a Functional Logic Block

These outputs are sampled by the other participating SCOPE octal, which generates a PSA for these patterns. A second circuit configuration that is well-suited for PSA/PRPG testing is shown in Figure 5.

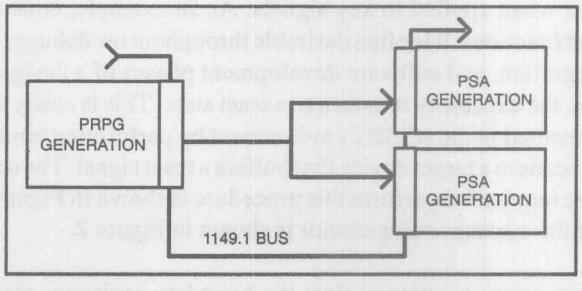


Figure 5. PSA/PRPG Generation for a Bus Configuration

While one of the several SCOPE octals connected to this bus is generating PRPG, the others can be placed in a PSA mode, and their subsequent signatures compared for consistency. A third candidate for PSA/PRPG is shown in Figure 6.

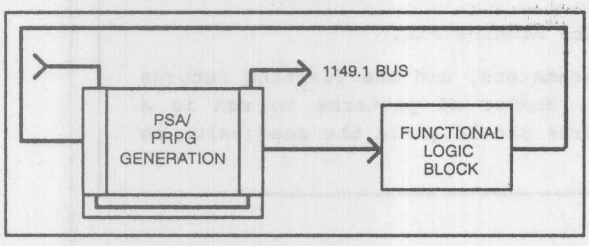


Figure 6. PSA/PRPG Generation for a Bus Configuration

In this case the SCOPE octal's outputs will be generating PRPG, and patterns will be created through the feedback and logic block path for sampling and calculating PSA on the octal's own inputs. The ASSET code to perform a PSA/PRPG test is relatively simple. Figure 7 lists a routine to perform this test on the circuit shown in Figure 8, which is an example of the configuration shown in Figure 6. Note the three steps involved in completing scan operations to the components: setting up the SCOPE PSA/PRPG commands, executing the test, and then reading back the results.

```
int mfm::led_data_psaprpg (int input_seed, int output_seed)

Name:
     int  mfm::led_data_psaprpg (int input_seed, int output_seed)
Description:
     Performs a PSA/PRPG test on the LED state machine PAL.

     Pseudo-random seeds are passed in as parameters, and the function returns
     the resulting signature as an integer. Number of patterns to run is a
     defined constant. For 8 bits, 256 patterns should yield the seed value as
     a result.

#define  NO_PSAPRPG_PATT       256
int psa_results
{
// set up
u34.scancn ();              // select SCOPE control register for LED SM reg
sc_scan  (IR);              // scan in instruction

u34.psaprpg ();             // set up to do PSA/PRPG
sc_scan  (DR);              // scan out to SCOPE  control register

u34.readbn ();              // select boundary, no observe/control
sc_scan(IR);                // scan in instruction

u34.enable();               // be sure outputs get turned on
u34.input  = input_seed;    // set up input and
u34.output = output_seed;   // output register seed values
sc_scan(DR);                // scan into boundary register

// run test
u34.runt ();                // select run SCOPE test instruction
sc_run (NO_PSAPRPG_PATT);   // run the test for 100 clocks

// retrieve and return results
u34.readbn ();              // select boundary, no observe/control
sc_scan  (IR);              // scan in instruction
sc_scan  (DR);              // scan out PSA/PRPG results

psa_results = u34.input;
return (psa_results);

}   // end led_data_psaprpg ()
```

Figure 7. Performing PSA/PRPG Tests Using ASSET and Scan

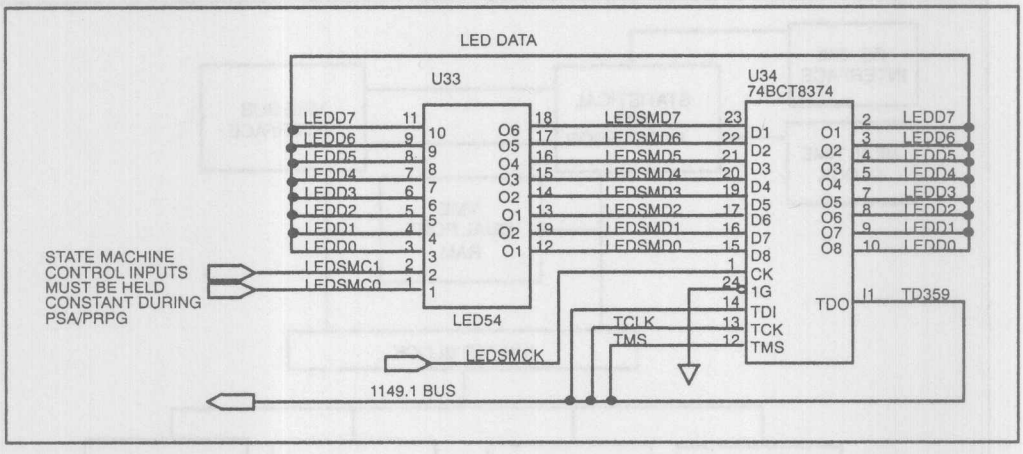


Figure 8. Register and Functional Circuitry for PSA/PRPG Test

Periodic BIT and Production Environment Testing

By incorporating a Texas Instruments Test Bus Controller chip into a design, an embedded processor can be used in addition to a remote host computer for controlling 1149.1 operations. In this manner, ASSET test routines that have been adapted to this environment can be applied at any time to complement the BIT software that is resident. The unique capabilities of the scan architecture thus permits a more complete and detailed testing of the hardware. ASSET software routines developed for debugging a design can be applied to other phases of the design cycle as well. Leveraging off the debug tests, automated testing for the production environment can quickly be achieved. Test patterns that may have been developed during the design phase, for an on-board ASIC or a board-level functional block, can still be applied and verified using scan. Additionally, unlike a 'bed-of-nails' or logic analyzer with test pattern generation, testing a design with ASSET functions is nonintrusive and requires no additional test equipment, other than the host 1149.1 controller.

How to Hold, Isolate, and Emulate a Functional Block

A powerful application of scan test techniques is the emulation of functional circuitry that is either not present, not operational, or under test. For instance, without disturbing the logic surrounding an ASIC, it can be thoroughly tested by applying test patterns at its input boundary cells, and sampling their results at the output boundary cells. These test patterns can often be directly taken from the design's original simulation test vectors, saving significant test development time. In the same manner, a functional block can be isolated and emulated at a board level, or a board at the system level. As an example, consider the block diagram shown in Figure 9.

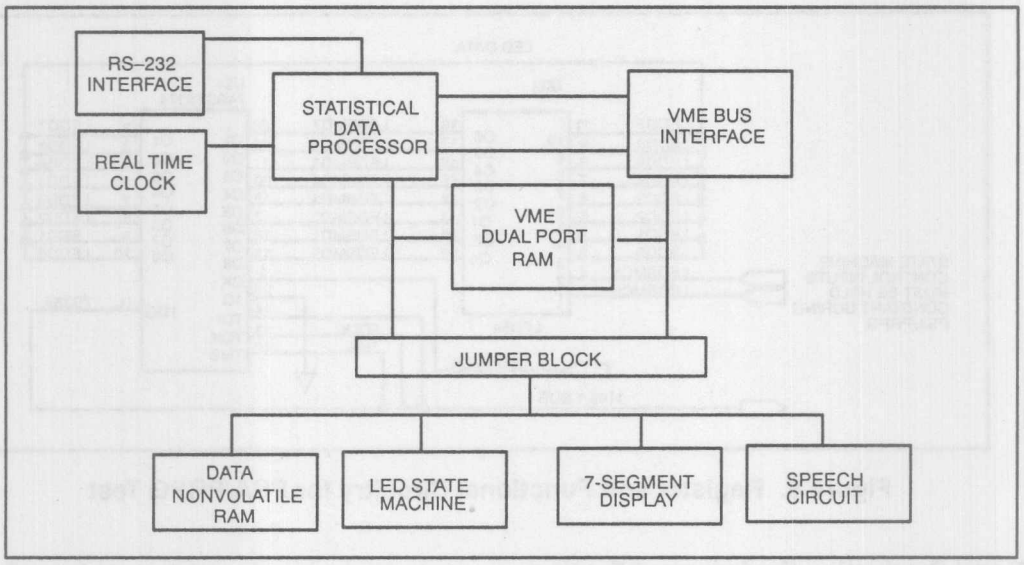


Figure 9. VME Multifunction Board

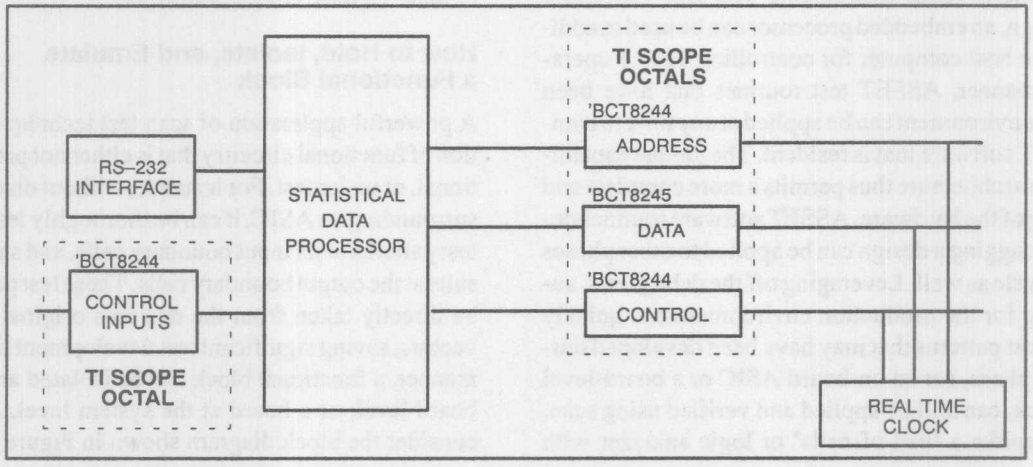


Figure 10. Processor Block Diagram

Figure 9 shows a circuit board for insertion into a VME backplane system. The on-board processor and the VME slave function each share and contend for the common resources (shown beneath the jumper block), as well as each having its own dedicated resources. Texas Instruments' 1149.1 components are embedded into each of the functional blocks to allow them to be individually tested. The scan chains for these components are designed so that each functional block can be controlled, then isolated, and finally, emulated. To demonstrate how an individual functional block is emulated, refer to the processor block diagram shown in Figure 10.

The address, data, and control lines are buffered as in any typical design, but TI SCOPE octals are substituted for their functional counterparts. Processor control inputs, such as reset, clock, and interrupts are buffered with SCOPE octals. Surrounding the processor with scannable devices is necessary when there is no direct control of it via scan. Including 1149.1 in the design of such chips, such as the TMS320C50, eases testability requirements for a system.

Holding the State of a Functional Block

The first step to emulating the processor's function is to control it by holding one or more of its control inputs, causing it to stop executing or otherwise becoming inactive. Holding the state of the processor in this manner will assure that while its address, data, and control outputs are being controlled by scan operations, it will not attempt to perform memory fetches or

proceed into an unknown state. Holding the processor's state can be done by controlling a wait or retry input, holding its reset active, inhibiting clock pulses, or causing it to execute from its internal memory while a control input is active.

Isolating the Functional Block From the System

Once a processor function is held in a stable condition, it then must be isolated from any effects that may cause it to exit the known state. Inputs such as interrupts should be controlled such that they do not interfere during the scan test mode. Outputs from the processor will be controlled by the address, data, and control scan buffers, overriding its normal outputs.

Processor Bus Emulation Using Scan and ASSET

Now that the processor has been held in a known state and isolated from external control inputs, the external board circuitry can be tested by emulating the processor. To emulate the processor, scan operations to control the address, data, and control lines are performed. Read/write strobes and chip select signals are generated in the proper sequence; for read operations, data is sampled when it is driven by the functional circuitry. An ASSET function to perform a write operation for the statistical data processor of Figure 9 (implemented with a TMS370C850 single chip microcomputer) is shown in Figure 11. Note that setup and hold times, which are emulated by performing separate scan operations, are easily met, since each complete scan will take on the order of microseconds instead of nanoseconds.

```
void mfm::sdp_write (UWORD adr, UWORD dat)

  Name:  mfm::sdp_write (UWORD adr, UWORD dat)
  Description:
  Writes the data passed to the address using the SDP scope octal buff-
  ers. Error checking is performed to make sure the address passed in is
  valid according to the SDP memory map. This is the core routine that is
  responsible for generating the proper bus timing for address, control
  and data lines via scanning

{
  int cspf = 1, ocf = 1, csh3 = 1;
  UWORD temp;
  if ((adr >= 0x10C0) && (adr <= 0x10FF)) cspf = 0;
  if ((adr >= 0x2000) && (adr <= 0x3FFF)) ocf = 0;
  if ((adr >= 0x8000) && (adr <= 0xFFFF)) csh3 = 0;
  if (cspf && ocf && csh3)  {     // 1 = Bad address encountered
     sys_error ("WBADADDRESS");
     return ;
     }
  u9.select_scan (SDP_BUD);       //  scan operations now begin
  u71.read ();                    //  point to address, data buffers'
  u69.read ();                    //  boundary scan cells without driving
  u70.read ();                    //  u71, u69 = address, u70 = data
  sc_scan (IR);                   //  scan out read scan instruction

  sdp_data_i = dat;               // set data outputs (u70)
  sdp_addr_i = adr;               // set address outputs  (u71, u69)
  u70.DIRAB;
  u70.enable ();                  // drive out data bus
  sc_scan (DR);                   //   perform the scan
                                  // set up octals to drive
  u66.readt ();                   // chip selects
  u71.cntl10 ();                  // address LSB
  u69.cntl10 ();                  // address MSB
  u70.readt ();                   // data bus
  sc_scan (IR);                   // scan out these instructions
                                  // set chip select, r/w low
  sdp_r_w_i = 0;                  // sdp r/w line set for write
  sdp_cspf_i = cspf;              // set appropriate chip select
  sdp_ocf_i = ocf;                //   low
  sdp_csh3_i = csh3;
  sc_scan (DR);                   // scan instruction to drive chip select
```

Continued ...

Figure 11. Emulating a Processor Write Function with ASSET (1 of 2)

```
    sdp_cspf_i = 1;              // bring chip select high
    sdp_ocf_i = 1;
    sdp_csh3_i = 1;
    sc_scan (DR);
    sdp_r_w_i = 1;               // bring read/write strobe high
    sc_scan (DR);
    u70.disable ();              // tri-state data bus
    sc_scan (DR);
    u66.cntl10 ();               // bypass ics
    u71.bypass ();
    u69.bypass ();
    u70.bypass ();
    sc_scan (IR);
}  // end sdp_write
```

Figure 11. Emulating a Processor Write Function with ASSET (2 of 2)

Using Scan Emulation to Test Isolated Functional Blocks

In addition to emulating embedded functions on a board, ASSET can be used with scan to isolate and test a board in a system backplane environment. Reference the VME slave function shown in Figure 9, then examine a more detailed view of this circuitry in Figure 12.

All of the control, address, and data signals required to perform VME transactions with resources on the board have been properly buffered through SCOPE octals so that they can be controlled in a test mode. In this manner, this functional block can be tested independently of the rest of the board. It is also possible to write ASSET code to control these signals to emulate transactions that would have normally originated from a VME master, while the remainder of the board is still in a functional mode. An example of the ASSET code to perform this is shown in Figure 13.

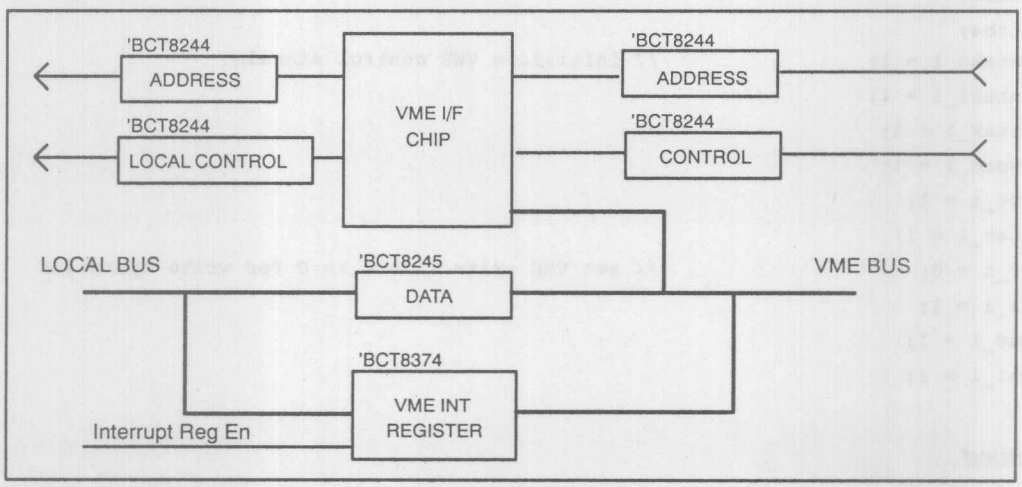


Figure 12. VME Bus Interface Block Diagram

```
void mfm::vme_write (UDWORD adr, UWORD dat, int i_o)
```

```
   Name:  void mfm::vme_write (UDWORD adr, UWORD dat, int i_o)
   Description:
   Writes from the VME buffers to the address passed in the data that is passed in. The
   parameter i_o distinguishes between VME I/O cycles and memory cycles.  Note that the
   octals that are used here are the ones that actually touch the VME backplane.
   This is the core routine that is responsible for generating the proper bus timing
   for address, control and data lines via scan.
```

```
{
adr_odd = 0x00000001;              // Performs write cycle from VME UDWORD
adr |= MFM_BASE_AD;                // maps the address into the MFM addressable space
adr_odd &= adr;                    // 1 for odd address
adr >>= 1;                         // align adr for logical assignment to vme_addr field
u9.select_scan (VME_BUD);
u82.read ();                       // point to components'
u83.read ();                       // boundary cells without driving
u85.read ();
u84.read ();
u89.read ();
u92.read ();
u86.read ();
u78.read ();
u80.sample ();
sc_scan (IR);                      // scan out these instructions
vme_addr_i = adr;                  // set address outputs
if (adr_odd) vme_odata_i = dat;    // assign data to odd or even
   else vme_edata_i = dat;         // data byte's buffer
u89.dirba;
u92.dirba;
vme_intako_i = 1;                  // initialize VME control signals
vme_intaki_i = 1;
vme_intak_i = 1;
vme_lword_i = 1;
vme_rst_i = 1;
vme_dtak_i = 1;
vme_wr_i = 0;                      // set VME write strobe to 0 for write operation
vme_as_i = 1;
vme_ds0_i = 1;
vme_ds1_i = 1;
```

Continued ...

Figure 13. Example of a VME Write Operation Performed with ASSET Scan Operations (1 of 3)

```
if (i_o == 1) vme_vam_i = 0x29; // set address mode lines for io
 else vme_vam_i = 0x39;         // else memory
sc_scan (DR);                   // scan this setup out
u82.setbyp ();                  // address nybble
u83.setbyp ();                  // address nybble
u85.setbyp ();                  // address nybble
u84.setbyp ();                  // address nybble
u89.readt ();                   // data byte
u92.readt ();                   // data byte
u86.readt ();                   // AM code, DS lines (= 1 now)
u78.readt ();                   // write line, other control signals
u80.sample ();                  // so we can read our dtack out
sc_scan (IR);                   // scan out these instructions
if (vme_dtak_o == 1)            // VME DTACK inverted from this signal name
  sys_error ("WBADOBSERVE");    // it should be a 1 now (vme_dtak_o = 0)
if (adr_odd) u92.enable ();     // set data enabled on bus
  else u89.enable ();
vme_ds0_i = ! adr_odd;          // set data strobes
vme_ds1_i = adr_odd;
sc_scan (DR);                   // scan out this data
sc_scan (DR);                   // a second scan will sample vme_dtak_o
if (vme_dtak_o == 0)            // VME DTACK should be 0 (vme_dtak_o = 1)
  sys_error ("WBADOBSERVE");    // to indicate VME state machine sampled
                                // data and is driving DTACK
vme_as_i = 1;                   // address strobe goes high
vme_ds0_i = 1;                  // pull data strobes high
vme_ds1_i = 1;
sc_scan (DR);                   // NOW
u89.disable ();                 // tri-state data bus
u92.disable ();
vme_wr_i = 1;                   // bring write strobe high
sc_scan (DR);                   // NOW
if (vme_dtak_o == 1)            // VME DTACK should be 1 (vme_dtak_o = 0)
  sys_error ("WBADOBSERVE");    // indicating state machine
                                // released DTACK
```

Continued ...

Figure 13. Example of a VME Write Operation Performed with ASSET Scan Operations (2 of 3)

```
  u82.bypass ();                    // set everything back to functional mode
  u83.bypass ();
  u85.bypass ();
  u84.bypass ();
  u89.bypass ();
  u92.bypass ();
  u86.bypass ();
  u78.bypass ();
  u80.bypass ();
  sc_scan (IR);                  // scan out bypass instructions

}  // end vme_write function
```

Figure 13. Example of a VME Write Operation Performed with ASSET Scan Operations (3 of 3)

Note that ASSET allows for the flexibility to refer to signals and nodes with synonyms that reflect their functionality or name on the corresponding schematic. Also note that the software generates control, address, and data signals to present a valid VME transaction to the rest of the circuit. While this procedure does not perform an "at-speed" test of the board, it does allow functional verification.

Considerations When Designing ASSET Programs

Avoiding Bus Clashes — Development of Constraint Files

With the ability to control individual signal outputs via scan comes the possibility that, through software, inadvertent clashes will occur from simultaneously driving more than one output on a node, possibly resulting in physical device damage. The ASSET system allows through software the ability to reduce and/or eliminate the possibility that device output clashes will occur. This ability is referred to as constraint checking. Constraint checking occurs before any scan operation, inhibiting a scan when an illegal condition is detected. A circuit should be examined thoroughly throughout the design and integration phases for possible constraint violations caused by scan operations, and the proper ASSET code incorporated as soon as possible, in order to prevent these anomalies.

Initialization of the ASSET Data Base

Another precaution that should be taken is initialization of the ASSET transmit buffer data base. To keep track of data that is to be sent or is received from the scan path, the host software platform that is executing ASSET scan instructions has two buffers for each component on the scan path: a transmit buffer and a receive buffer. Each transmit buffer holds the exact data that is to be transmitted into the scan path on the subsequent data scan to a particular component. Each receive buffer holds the exact data received from the scan path on the previous data scan from a component. As is the case with all data structures, these buffers contain unknown values until they are initialized. In writing ASSET code for test programs or in using the ASSET debugger, it is possible to change single bits at the output of a component. At the same time, all the other outputs of that component are driven to a logic level determined by the data in its transmit buffer. In many cases, the other bits may be key functions (such as resets, enable lines, clocks, etc.) that would cause a circuit failure (in terms of the test being attempted) if they were set to the wrong value. To minimize the possibility of such an incident, the transmit data base for each component should be initialized to a value that will cause the least disturbance to circuit behavior. It should be kept in mind that these initialized values may be changed (through single-bit or full-word instructions) as individual tests use the component. Upon completion, these tests should return all signals to their safe logic level. To ensure that all signals are initialized to a safe level, a designer should examine the target system, identify all signals that have a preferred value if inadvertently turned on, and incorporate these into an initialization function. This function should be called upon entry into the ASSET code. An example of an initialization routine is shown in Figure 14. Note that the initialization of signals consists of a simple assignment statement.

```
void mfm::scan_tx_init (void)

 Name:    mfm::scan_tx_init (void)
 Description:
             Initializes the ASSET data base to a known state.

   Note that this is a reset of the ASSET SCAN transmit data base -
   this does not actually issue a MFM master reset, nor actually
   perform a scan operation.

  UWORD temp;

//  TDI ------------------------------------> TDO
//  boundary of BCT8244:  1G_, 2G_, INPUT (7:0), OUTPUT (7:0)
//  boundary of BCT8245:  DIR,  G_,     A (7:0),      B (7:0)
//  boundary of BCT8373:    C, OC_, INPUT (7:0), OUTPUT (7:0)
//  boundary of BCT8374:  CLK, OC_, INPUT (7:0), OUTPUT (7:0)
//  Refer directly to the schematics to understand why the
    initialization values are as they are.
//  Note that it doesn't matter what state the input buffer is
    initialized to since it must be read before any data in it is valid.
//  Concerns here are the control inputs (make certain they are turned
     off!) and the output registers (set to harmless values).

// VME bud
    u78.output = 0xFF;              // VME bus control signals
    u78.aux    = 0x00;              // BCT8244 enables
    u86.output = 0x03;              // VME bus control signals
    u86.aux    = 0x00;              // BCT8244 enables
    u80.output = 0xFF;              // VSAM, VME CS control signals
    u80.aux    = 0x00;              // BCT8244 enables
// SDP bud
    u72.output = 0xFF;              // SDP control inputs
    u72.aux    = 0x00;              // BCT8244 enables
    u66.output = 0xFF;              // SDP output control signals
    u66.aux    = 0x00;              // BCT8244 enables
    u18.output = 0xFF;              // JB chip select buffer
    u18.aux    = 0x00;              // BCT8244 enables
// SPCH bud
    u25.output = 0xFF;              // JB chip selects and clk buffers
    u25.aux    = 0x00;              // BCT8244 enables
    u51.output = 0xFF;              // Speech last address register
    u51.g__    = 0x00;              // BCT8245 enable
    u55.output = 0xFF;              // Speech first address register
    u55.g__    = 0x00;              // BCT8245 enable
    u62.output = 0xFF;              // Speech command to proc buffer
    u62.aux    = 0x00;              // BCT8244 enables
    u45.output = 0x54;              // Speech state machine buffer
    u45.aux    = 0x00;              // BCT8244 enables

  }  // end scan_tx_init
```

Figure 14. Scan Transmit Buffer Initialization Routine

Scan Path Configurations for Performing PSA/PRPG Operations

While it is possible to implement a design with more than one scan path, there is software overhead involved in switching from one scan path to another. This overhead is dependent on the host controller's timing and other events that may not be related to 1149.1 operation. When performing PSA/PRPG tests on a circuit, it is required that a start command be issued simultaneously to all participating components. It is recommended that all components to be used for PSA/PRPG testing be located on the same scan path. This will guarantee that scan commands to the components under test are synchronous, and will ensure consistent results from the PSA/PRPG testing.

Conclusions

Decreasing circuit geometries, implemented with improved assembly techniques and increased integration of functionality onto single chips, demands state-of-the-art methods of testing a design. A serial scan bus provides an excellent means for controlling and observing nodes that are inaccessible using conventional test methods. The introduction of scan architectures into a design facilitates an entirely new approach to debugging and integration. Some of the many benefits are:

- Reduced dependence on external test equipment. Logic analyzers, oscilloscopes, digital multimeters, logic probes, and specialized test equipment can be replaced by scan functionality.
- An improved diagnostics and testability approach that is closely integrated to the functional circuit design process.
- Reduced maintainability costs over the life of a design through improved fault isolation.
- Recycling of test software for each phase of the life of the design.
- Easily updated test strategies. Improvements or modifications to a design can be reflected in the test procedures through software-only changes.

Hardware-Based Extensions to the JTAG Architecture

by Lee Whetsel and Greg Young

This paper was presented at the ATE and Instrumentation Conference West, 1990.

IEEE 1149.1 Overview

The 1149.1 IEEE standard provides an IC-level test framework consisting of a four-wire Test Access Port (TAP) controller and related scan path architecture, as shown in Figure 1. The TAP controller receives external control input via Test Clock (TCK) and Test Mode Select (TMS) signals, and outputs control signals to the internal scan paths.

The scan path architecture consists of a single serial instruction register and two or more serial data registers. The two required data registers are a boundary scan register and a scan bypass register bit. The instruction and data registers are connected in parallel between a serial Test Data Input (TDI) signal and serial Test Data Output (TDO) signal. The TDI input is connected directly to the serial inputs of the instruction and data registers. The TDO output is connected via multiplexer 1 to either the serial output of the instruction or data registers. The selection control for multiplexer 1 comes from the TAP controller. Since multiple data registers can be used, multiplexer 2 is required to route a selected data register's serial output into multiplexer 1 to drive the TDO output. The selection control for multiplexer 2 comes from the instruction register.

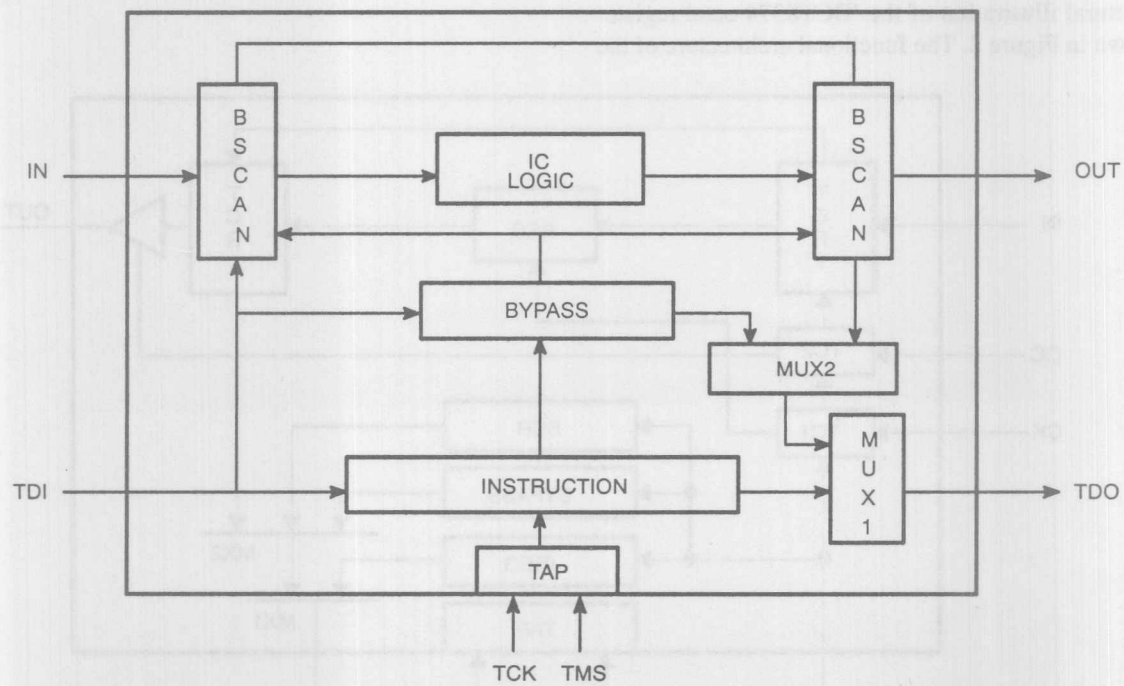


Figure 1. 1149.1 Architecture

When boundary testing is not being performed, the boundary scan register is transparent, allowing input and output signals to pass to and from the IC logic. However, during boundary testing, the boundary scan register disables the normal flow of input and output signals to allow the boundary signals of the IC to be controlled and observed via scan operations. The 1149.1 standard details how multiple ICs incorporating this scan architecture can operate together to perform external wiring interconnect tests between the boundaries of ICs in a circuit.

SCOPE Octal ICs

The SCOPE octals are a first in a series of standard components to be offered by TI that blend functionality with embedded board-level testing features [1,2]. The initial products include an octal register type 'BCT8374, latch type 'BCT8373, buffer type 'BCT8244, and transceiver type 'BCT8245. Along with the normal function pins associated with each part, four pins are added to support the 1149.1 test bus interface signals, TDI, TMS, TCK, and TDO. These devices can be substituted for their nontesting counterparts in a variety of board-level design applications such as: pipeline registers, board I/O buffers, address and data buffers/transceivers, and finite state machine designs.

An architectural illustration of the 'BCT8374 octal register type is shown in Figure 2. The functional architecture of the 'BCT8374 octal consists of an 8-bit register (REG), eight data inputs (IN), eight data outputs (OUT), a clock input (CK), and a tristate output control input (OC). The 1149.1 architecture consists of a Test Access Port (TAP) controller, an instruction register (IREG), and a data register section. The data register section consists of a bypass register, a boundary control register (BCR), and a boundary scan register. The boundary scan register consists of SCOPE Test Cells 1 and 2 (TC1, TC2) coupled to the CK and OC inputs, SCOPE Test Cell Register 1 (TCR1) coupled to the IN inputs, and SCOPE Test Cell Register 2 (TCR2) coupled to the OUT outputs. The other SCOPE octals have a similar 1149.1 test architecture placed around buffer, transceiver, and latch functions.

The TAP controller receives external input from the TMS and TCK signals, and outputs internal control to either the IREG or a selected data register to cause a shift operation to occur from the TDI input to the TDO output. The IREG is used to store a test instruction to be executed by the IC. The bypass register is used to shorten the scan path length through the IC to a single bit during data register operations. The BCR is used to store boundary configuration control bits to extend the test capabilities of the boundary scan register. The boundary scan register provides the mandatory test features required for 1149.1 compatibility as well as extended test features developed for SCOPE products.

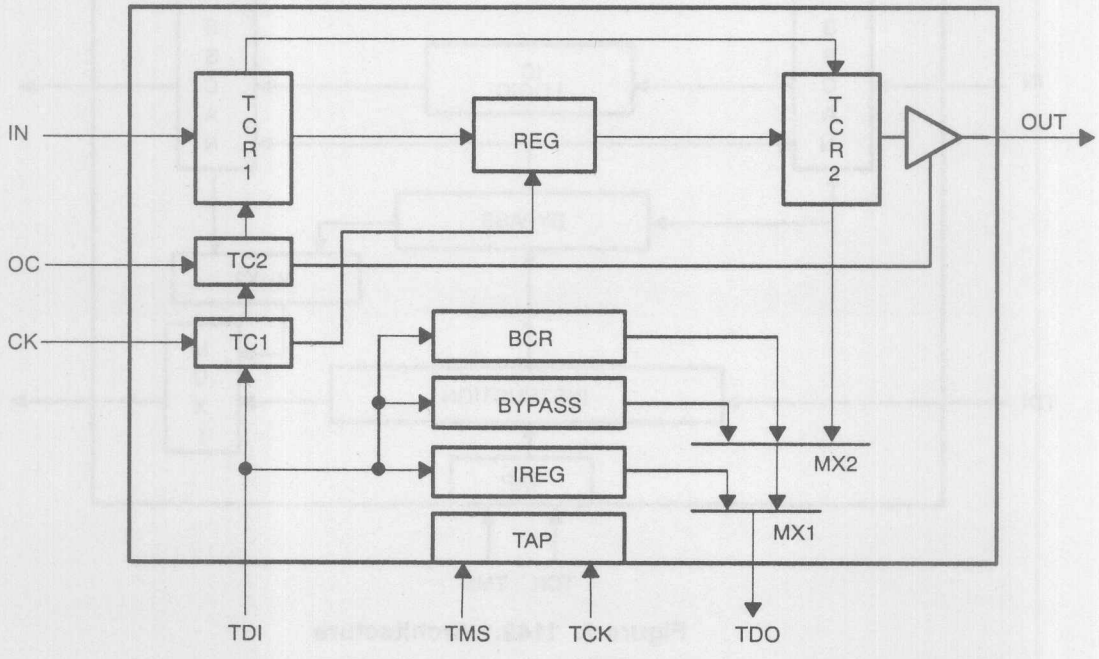


Figure 2. Scope Octal Register

Normal Mode Operation

During normal operation, the boundary scan register is transparent, allowing input and output signals to pass freely through the test cells, enabling the device to perform its intended function. While in normal operation, the TAP can receive control from the TMS and TCK inputs to shift data through the device from the TDI input to the TDO output. Three test instructions can be executed while the device is in normal mode: 1149.1 SAMPLE and BYPASS instructions and a SCOPE cell self-test instruction. The SAMPLE instruction allows the data flowing through the boundary to be sampled, then shifted out for inspection. The BYPASS instruction selects the bypass register to be shifted during data register scan operations, reducing the scan path length through the device to 1 bit. The self-test instruction executes a self check of each SCOPE cell in the boundary scan register. While the SAMPLE instruction at first may appear very attractive, the user of these and other 1149.1-compatible devices must know when to sample in order to obtain meaningful data.

Test Mode Operation

When placed in an off-line test mode, the normal operation of the SCOPE octal is inhibited. In test mode, instructions can be input to the device to perform all mandatory 1149.1 instructions as well as an extended set of test instructions designed for SCOPE products. The test-mode instructions incorporated into all SCOPE octals are described below. Prior to loading these test-mode instructions, the boundary scan path should be set so that a desired first test control pattern is applied to the REG inputs, tristate buffers, and device outputs (OUT). This procedure ensures that the device will be in a known state when the test mode is entered.

When an 1149.1 external or internal boundary test (EXTEST or INTEST) instruction is input to the device, the boundary register (TC1, TC2, TCR1, TCR2) is set to allow simultaneous observation of signals input to the test cells and control of signals output from the test cells. Simultaneous control and observation is achieved by the design of the test cells used to construct the boundary register. Each test cell contains two memories (flip-flops), one to observe input data and the other to control output data.

During 1149.1 EXTEST or INTEST, the TAP receives external input to cause the boundary register to capture data on the CK, OC, and IN inputs as well as on the internal REG outputs. The input memories maintain a desired control output logic state to internal as well as external logic inputs. For example, TC1 maintains a desired control input to the OUT tristate buffers, while also capturing data input on the OC pin. After the data is captured, the TAP receives additional input to shift the stored data out via the TDO outputs. While captured data is shifted out, the next test control pattern to be output from the boundary register is shifted in via the TDI input. The boundary register outputs remain in their present state during the shift operation. At the end of the shift operation, the TAP receives further input to cause the next test control pattern to be output from the boundary register. This process of capturing input data, shifting the boundary register to extract stored data and to load new test control data, followed by the application of the new test control data from the boundary register outputs, is repeated as required to perform a particular EXTEST or INTEST operation.

Test Extensions

To support extended testing capabilities, additional instructions, control logic and test pins are defined in the SCOPE architecture. Some of the SCOPE test extensions are included in the octal devices. The benefits of developing an extended test architecture is that all TI parts will share a consistent test instruction set, compatible testing modes, and reduced complexity in the development of test and maintenance software tools. The following paragraphs describe some of the SCOPE instructions that are included in the octal devices.

TRIBYP Instruction

When a SCOPE TRISTATE outputs and BYPASS (TRIBYP) instruction is input to the device, the outputs are placed in a high-impedance state and the bypass register is selected. This instruction is designed primarily to facilitate a blend of in-circuit and boundary scan testing. By disabling the outputs of the device, an in-circuit tester can drive the inputs of another device coupled to the outputs of the octals without damaging the octal's output buffers. While this instruction is in effect, the bypass register is selected to provide a minimum data register scan length through the device.

SETBYP Instruction

When a SCOPE SET outputs and BYPASS (SETBYP) instruction is input to the device, the boundary outputs are set to a prescanned combination of logic 1's and 0's and the bypass register is selected. This instruction allows the boundary test cells to output a prescanned control pattern to the REG inputs, tristate buffers, and device outputs (OUT). The SETBYP instruction allows placing the octal device in a preferred static input and output state while testing of neighboring components is being performed. While this instruction is in effect, the bypass register is selected to provide a minimum data-register scan length through the device.

READB Instruction

When a SCOPE Read Boundary (READB) instruction is input to the device, the contents of the boundary register can be shifted out. This instruction differs from the 1149.1 EXTEST or SAMPLE instruction in that the capture operation that normally occurs in the TAP's data-register capture state is replaced with a data-register hold operation. The data-register hold operation causes the boundary register test cells to capture their present state instead of the data values input to the test cells. This instruction is primarily used to allow a signature that has been collected in the boundary register to be shifted out for inspection.

RUNT Instruction

To support a boundary Built-In Self-Test (BIST) approach, a Run Test (RUNT) instruction was developed for SCOPE devices. RUNT is a generic instruction that executes the boundary BIST operation setup by control bits programmed in the BCR, shown in Figure 2. The BCR control-bit settings must be set up via a scan operation prior to loading the RUNT instruction. All RUNT programmations execute while the TAP controller is in the Run Test/Idle state. The length of a particular RUNT test operation is controlled by the number of TCK inputs applied while the TAP is in the Run Test/Idle state.

The four programmations of the RUNT instruction implemented in the octals include: 16-bit Parallel Signature Analysis (PSA) of the IN inputs, 16-bit Pseudo-Random Pattern Generation (PRPG) from the OUT outputs, simultaneous PSA of IN inputs and PRPG of OUT outputs, and simultaneous SAMPLE of IN inputs and TOGGLE of OUT outputs.

During the 16-bit PSA RUNT programmation, the 8-bit TCR1 and TCR2 boundary sections are linked together to form a single 16-bit Linear Feedback Shift Register (LFSR). The parallel inputs to TCR1 are enabled to accept data from the IN bus and the parallel inputs to TCR2 are disabled. In this configuration TCR2 acts as an 8-bit LFSR extension to TCR1. During test, the parallel inputs from the IN bus are compressed into the 16-bit LFSR on the rising edge of TCK. Linking TCR1 to TCR2 allows the SCOPE octal to receive an extended sequence of 8-bit patterns from the IN bus. At the end of the PSA operation, the 16-bit signature can be shifted out of TCR1 and TCR2 for inspection. While TCR1 and TCR2 are collecting the signature, the outputs of TC1 and TC2 remain in their present state. TC2 can be set to enable or disable the OUT buffers during the test.

During the 16-bit PRPG RUNT programmation, the 8-bit TCR1 and TCR2 boundary sections are linked together to form a 16-bit LFSR as described in the 16-bit PSA test. During the 16-bit PRPG test operation, both parallel inputs to TCR1 and TCR2 are disabled so that both act only as LFSRs. During test, the parallel output from TCR2 drives pseudorandom patterns to the OUT bus on each falling edge of TCK. By linking TCR1 and TCR2 together, an extended set of pseudorandom pattern sequences is produced. Since the width of the OUT bus is 8 bits, individual patterns will be repeated during every 256 pattern output sequence. However, the test circuit will produce 256 sets of unique 256 pattern output sequences. During this test, TC2 must be set to enable the OUT buffers.

During the simultaneous PSA and PRPG RUNT programmation, TCR1 and TCR2 operate as two separate 8-bit LFSRs. The parallel inputs to TCR1 are enabled to accept data from the IN bus and the parallel inputs to TCR2 are disabled. During test, TCR2 outputs pseudorandom patterns to the OUT bus on the falling edge of TCK, and TCR1 compresses input data from the IN bus on the rising edge of TCK. Combinational logic residing in the external path between the OUT and IN buses can be quickly tested using the RUNT instruction. During this test, TC2 must be set to enable the OUT buffers.

During the simultaneous Sample Inputs/Toggle Outputs RUNT programmation, TCR2 outputs alternating data patterns to the OUT bus on the falling edge of TCK, and TCR1 accepts data input from the IN bus on the rising edge of TCK. By adjusting the frequency of TCK, this test can be used to measure propagation delays through external logic residing between the OUT and IN buses. During this test, TC2 must be set to enable the OUT buffers.

SCOPE Octal Applications

A typical statement from a design engineer/manager may be "These parts are interesting, but they are more expensive than ordinary registers and buffers. Where is the payback?" As with any new product, the historical data required to substantiate any cost savings claims is not available, and even if it were, it would be based on a specific benchmark example application, not in general. Functionally, these parts offer no advantages over existing parts. However, if an interest exists in reducing the cost associated with product manufacturing and test, these parts deserve a second look. The following example illustrates how the octals can be used to improve test and diagnostics in a production environment.

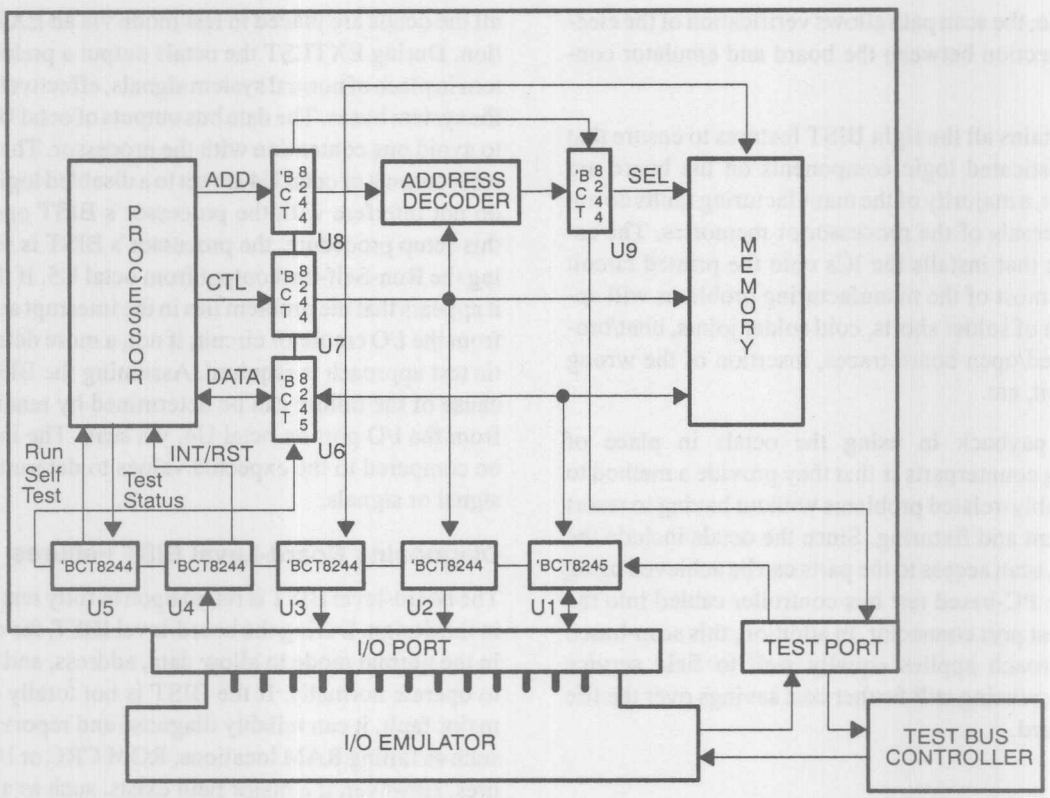


Figure 3. Microprocessor Board Design

A typical microprocessor board design is shown in Figure 3. The design includes a processor, address decoder, memory, buffering logic, and an input/output port. The processor has address, control, data, and interrupt buses that are routed to onboard memory and off-board peripherals via the I/O port. The address decoder consists of combinational logic and decoders. The memory consists of ROM and RAM. The buffering logic consists of SCOPE octal 'BCT8244 buffers and 'BCT8245 transceivers. These octals are placed on the busing paths to provide functional buffering and embedded testing features between the major circuits in the design, as well as external I/O circuitry.

Proper steps have been taken in the design to ensure the board can be manufactured in a cost-effective manner. While several processors meet the functional requirements established for the design, one has been selected that includes an embedded BIST capability that can be initiated at power up or by a Run-Self-Test pin to verify the processor's internal logic. The processor includes a Test Status output pin that outputs a particular sequence to indicate whether the internal BIST operation passed or failed. Test pins Run-Self-Test and Test Status are coupled to an extra octal (U5) to allow the processor BIST to be initiated and the results to be monitored via scan. If the processor BIST fails, no further testing occurs until the failure is diagnosed and repaired.

In addition to the processor's internal BIST, memory locations in the onboard ROM are allocated to support an overall board-level BIST routine. If the processor passes the internal BIST test, it will proceed to the next test step of executing the board-level BIST routine. This ROM-resident board-level BIST is designed to cause the processor to execute an effective pattern test of the onboard RAM, a Cyclic Redundancy Check (CRC) of the contents of the onboard ROM, and I/O operations to external circuits attached to the I/O port. Since this test follows the processor BIST, it is executed at power up and can also be retriggered by invoking the processor BIST using the Run-Self-Test input pin.

If the board-level BIST passes, the board is considered good and requires no further internal functional testing. A pass or fail condition is recognized by reading a BIST status code written into a particular address location of the external I/O emulation circuitry. The I/O emulation circuitry is inserted into the I/O port connector to allow the processor to exercise off-board memory accesses. The I/O emulator includes an 1149.1 test bus to allow scan access of the board-level BIST

status code. Also, the scan path allows verification of the electrical interconnection between the board and emulator connectors.

The design contains all the right BIST features to ensure that the most sophisticated logic components on the board are tested. However, a majority of the manufacturing faults do not involve the internals of the processor or memories. The assembly process that installs the ICs onto the printed circuit board is where most of the manufacturing problems will appear in the form of solder shorts, cold solder joints, bent/broken pins, shorted/open board traces, insertion of the wrong logic component, etc.

Some of the payback in using the octals in place of their nontesting counterparts is that they provide a method to diagnose assembly-related problems without having to resort to test equipment and fixturing. Since the octals include the 1149.1 test bus, scan access to the parts can be achieved using a cost-effective PC-based test bus controller cabled into the board via the test port connector. In addition, this scan-based diagnostic approach applies equally well to field service applications, producing still further cost savings over the life cycle of the board.

Scan-Based Test and Diagnostics

As mentioned, the design includes two levels of BIST — a processor self check and a board-level self check. If both BIST operations are successful, the board is good and requires no further testing. However, if one of the BIST approaches fail, the 1149.1 test bus controller can scan the octals into their test mode and perform diagnostic tests to identify the problem that caused the BIST to fail. The following is an example design that illustrates how the octals can be used to test for failures at the board level.

Diagnosing Processor BIST Failures

Since the processor's BIST verifies its internal circuitry, a failure output sequence on the Test Status pin indicates this failure. Upon receiving this failure sequence, the processor should be replaced. However, if the processor's BIST fails to output the correct pass or fail sequence, the problem may not be in the processor. In this case, further investigation is required to see if the BIST is being affected by external conditions.

As shown in Figure 3, octal U4 inputs reset and interrupt signals to the processor from the off-board I/O emulator. If one or more of these signals is faulty, the operation of the processor's BIST could be affected. To determine if this is occurring, all the octals are placed in test mode via an EXTEST instruction. During EXTEST the octals output a preloaded test pattern in place of normal system signals, effectively partitioning the system buses. The data bus outputs of octal U6 are tristated to avoid bus contention with the processor. The interrupt and reset outputs of octal U4 are set to a disabled logic state so they do not interfere with the processor's BIST operation. After this setup procedure, the processor's BIST is retriggered using the Run-Self-Test output from octal U5. If the test passes, it appears that the problem lies in the interrupt and reset inputs from the I/O emulator circuit; if not, a more detailed diagnostic test approach is required. Assuming the BIST passes, the cause of the failure can be determined by reading the inputs from the I/O port on octal U4, via scan. The inputs read can be compared to the expected values to determine the failing signal or signals.

Diagnosing Board-Level BIST Failures

The board-level BIST is relied upon to fully test each function in the design. During the board-level BIST, the octals must be in the normal mode to allow data, address, and control buses to operate normally. If the BIST is not totally disabled by a major fault, it can reliably diagnose and report test problems such as failing RAM locations, ROM CRC or I/O access failures. However, if a major fault exists, such as a short or open condition on the address, data, or control bus, the processor will be unable to access the BIST code in ROM and the test will be disabled. In the event the board-level BIST is disabled, the following series of diagnostic tests can be executed by the octals.

Testing for Shorts and Opens Between Octals

If the board-level BIST is disabled, the octals can be configured to test for board trace shorts and opens. To initiate the test, the octals are loaded with the EXTEST instruction. The EXTEST instruction places the octals in test mode and allows the external test bus controller to input control to capture data appearing on the octal inputs and to control data from the octal outputs. During this test, the octals not involved with shorts and opens testing should output a safe static pattern to the logic they drive. In the case of U5, the reset output should be set to force the processor into a reset state during test.

Using this approach, wiring interconnects between the octals can be tested for shorts and opens. Octals U8 and U3 test the address bus for shorts and opens. Octals U7 and U2 test the control bus for shorts and opens. Octals U6 and U1 test the data bus for shorts and opens. While the onboard buses are being tested, a shorts and open test can also be performed between the I/O port and the emulator circuit, using octals U1, U2, U3, and U4. If any faults are detected using this approach,

they should be corrected and the board-level BIST repeated to determine if the detected faults were the ones disabling the BIST operation. If no faults are detected with this test, the octals can be configured to test for other faults.

Testing for Opens Between Octals and Memory/Address Decoder

The interconnect tests between the octals only verify that the board traces between the octals are intact and that no shorts exists. However, an open trace condition could still exist between the octals and address decoder and/or memory. To test the integrity of the interconnections to the address decoder and memory, the octals can be set up to emulate processor write and read operations. During this test, octals U8, U7, and U6 are loaded with the EXTEST instruction, octals U1, U2, U3, U4, and U5 are loaded with the SETBYP instruction, and octal U9 remains in the normal mode by loading a BYPASS instruction. Octal U9 remains in the normal mode to allow transferring the select outputs from the address decoder to the memory. During this test, all octals containing SETBYP instruction output a safe static pattern to the logic they drive.

By placing octals U8, U7, and U6 in the EXTEST boundary scan mode, the address, control and data buses can be operated by the external test bus controller to perform write and read memory accesses. If data can be transferred to and from all the memory address locations, no open trace or other failure conditions exist between the memory/address decoder and octals. However, if all or some of the data cannot be transferred to and from memory, two failure possibilities exist. The first possibility is that the address decoder is faulty. The second possibility is that an open trace may exist on the address decoder's select outputs.

To test for the second possibility, octal U9 is loaded with the EXTEST instruction to allow the select outputs to be controlled by the test pattern scanned into octal U9. This eliminates the address decoder from the test. By repeating the memory write/read test with U9 outputting the memory select signals in place of the address decoder, it is possible to determine if an open trace exists in the select signals. If the memory access test passes, the busing paths between the octals and memory are intact and the problem lies in the address decoder. If the memory access test fails, one or more open traces exist between the octals and memory. In the event the test fails, intelligent test patterns can be used to help diagnose where the open trace conditions exist.

Boundary Testing the Address Decoder

After verifying that opens do not exist on the data, control, select, or address buses, the address decoder needs to be thoroughly tested. To verify the logic inside the address decoder, octals U9, U8, and U7 are loaded with the EXTEST instruction. The other octals are loaded with the SETBYP instruction to allow them to output a safe static pattern while the address decoder is tested. During EXTEST, the external test bus controller inputs control to allow octals U8 and U7 to drive the inputs of the address decoder, and octal U9 to read the response outputs from the address decoder. The responses read by octal U9 are compared to expected values to see if the decoder is operating properly. If proper operation is determined, a timing-related problem could still exist and cause the address decoder to fail during BIST testing. The RUNT instruction is used to test for timing-related failures.

At-Speed Testing the Address Decoder

At-speed testing of the address decoder can be achieved by two of the RUNT programmations offered in the octals — the PSA/PRPG or the Toggle/Sample. During either of these RUNT tests, octals U7, U6, U5, U4, U3, U2, and U1 should contain the SETBYP instruction to allow their outputs to remain set at a safe static value. Also, the outputs of U9 should be set tristate during these tests to guard against the memory receiving multiple select inputs. Pull-up resistors placed on the outputs of U9 disable the select signals while they are tristate.

If the PSA/PRPG RUNT programmation is used, octal U8 outputs a pseudorandom pattern to the inputs of the address decoder on the falling edge of TCK, and octal U9 compresses the data from the address decoder into a signature on the rising edge of TCK. The TCK frequency determines the speed at which the test operates. After a predetermined number of TCKs, the signature collected in U9 is scanned out and compared to an expected signature. If the signature collected matches the expected value, the test passed; otherwise, the address decoder has a timing-related failure. If a failing signature is obtained, the Toggle/Sample RUNT programmation can be used to further diagnose the timing problem.

The Toggle/Sample RUNT programmation is primarily used for propagation delay testing. If the PSA/PRPG programmation is changed to the Toggle/Sample programmation, octal U8 outputs a toggling pattern to the inputs of the address decoder on the falling edge of TCK, and octal U9 samples the data output from the address decoder on the rising edge of TCK. The TCK frequency determines the speed at which the test operates. After each Toggle/Sample operation, the sampled outputs are scanned out of U9 for inspection, the next address value to toggle is scanned into U8, and the test is repeated. The advantage of this test over the PSA/PRPG test is that it provides deterministic delay testing of the address decoder. In other words, instead of indicating a failing signature,

this test identifies each output signal that fails to transfer through the address decoder between the falling and rising edge of TCK.

SCOPE Octal Application Summary

The described test approaches are made possible by substituting SCOPE octals in place of standard buffers and transceivers in a board design. The capability to provide the types of embedded scan-based tests described in this example can prove to be a cost-effective alternative to existing board design and manufacturing approaches. It is important to note that all the test features described are achieved by only having scannable octals in the board design. If the processor and memories were also scannable, the board could be tested even more thoroughly.

Conclusion

The 1149.1 standard provides the framework for a structured test approach. Devices described in this paper implement the boundary test functions required by the standard. In addition, the devices contain test extensions to improve their ability to test for timing-related faults that are not detectable using boundary scan techniques alone. Additional SCOPE products are being developed to provide additional test features to support board-level design for testability.

References

[1] *SCOPE Octal Data Sheets*, Local Texas Instruments Sales Office.

[2] A. Halliday, G. Young, and A. Crouch. *Prototype Testing Simplified by Scannable Buffers and Latches,*. Proceedings IEEE International Test Conference, 1989.

IEEE 1149.1 Use in Design for Verification and Testability at Texas Instruments

by Adam Cron

Abstract

Texas Instruments' hierarchical testability efforts have produced several new products aimed at standardization and cost reduction of IC and system test and debug[1]. These new products are compatible with the IEEE 1149.1 scan protocols and standards.[2,3] Included are ASIC cells, standard interface ICs, a bus master IC, a controller interface board for IBM compatibles, a high-speed scan interface, and software to control the scan bus. In this paper, I will introduce these products and explain a little about each. A detailed evaluation of some trade-offs to be looked at when using the IEEE 1149.1 standard for ASIC testability is also presented. Future products are currently under consideration.

Introduction

Let's take a quick look at design and manufacturing on the cutting edge. Your system is to perform supercomputing activities in engineering environments. To keep your boards from overheating, a complex liquid thermal transfer system is in place. Each board is a multilayer, controlled impedance, conformally coated unit comprised of ECL and BiCMOS fine-pitch surface-mounted components in the 10K to 100K gate range. Interconnects are kept to a minimum to improve system performance and reliability. Your task is to diagnose a fault in this system without totally disassembling it.

Not all system testability- and fault-diagnosis problems will be this complex. But each new design will have its own special set of constraints.

System densities are increasing and manufacturing processes are improving to the point where "in-circuit" probing of components is no longer feasible. Higher frequency circuits react with surrounding devices, dictating that they must be tested from within their functional environments. The complexity of today's electronics leaves little room for conventional ad-hoc testability features or design verification "hooks". The time for standardization of a test approach is now.

Inception

The Defense Systems and Equipment Group (DSEG) of Texas Instruments (TI) has always worked on the leading edge of many technologies. As circuit densities increased, and new manufacturing techniques permitted higher package counts, the need to integrate IC, board, and system test and verification circuitry within each of these respective levels of a design became necessary. From a system standpoint, it became obvious that whatever the method used to achieve this testability, a standard approach had to be taken. A method by which all levels of a system could be tested–from the pre-packaged bar to the system in field. The modular approach of this test and verification system became known as Hierarchical Testability.

Momentum

In 1986, Texas Instruments became an active member in the Joint Test Action Group (JTAG). This international organization, composed of semiconductor users and vendors from Europe and North America, had as its charter to define a testability strategy that would solve many of the problems design and manufacturing groups were facing. The solution of choice was boundary scan—a system whereby each I/O pin of a device may be observed or controlled via a four-wire scan interface and associated protocols. This system, now known as the JTAG or IEEE 1149.1 scan bus, enables large modules to be tested and evaluated from the IC-level up, while configured in their normal operating environments.

Solution

Texas Instruments has been a strong leader and proponent of a system solution to testability. TI's solution is an IEEE 1149.1-compatible one. But it goes further than simple

boundary scan. The System Controllability, Observability, and Partitioning Environment (SCOPE)[4] architecture and associated family of products has given the designer new tools to achieve the desired results of easy Design-For-Testability (DFT), as well as verification and debug features. Via the scan path, a virtual pattern generator/logic analyzer is created. This pattern generator/logic analyzer can, via the scan path, control and observe the inputs and outputs of a device.

To Probe Further

Parts of the Whole

There are many components that make up the SCOPE family of products. Each part works well within the architecture to help solve many design test and verification problems. Beginning with the most basic element, each component will be described and its many uses touched upon.

SCOPE Cells: The basic building block of the IEEE 1149.1 boundary scan architecture is the scan cell. The scan cell can control or observe a net in an ASIC. The version of this cell provided in TI SCOPE workstation libraries for use in ASIC Standard Cell and Gate Array designs is both compact and very versatile. By placing the scan cell between the I/O buffer and core logic, complete isolation of the device is achieved–a desirable goal for device and system testability.

There are currently 14 SCOPE Cells in the TI workstation libraries[5]. They range in functionality from simple controllability/observability cells to more complex cells used in Built-In Self-Test (BIST) applications. All cells offer the ability to control and observe a node at the same time. This feature can be used to the test engineer's advantage. He can execute internal device tests on the ASIC while performing exhaustive manufacturing tests on the board or system. This saves valuable test execution time. The advanced BIST functions built into some SCOPE Cells can be used to generate a pseudo-random set of patterns from each cell, or develop a signature within several cells based on the data observed by those cells. These are powerful functions and timesaving tools for the design/test engineer.

To enable communications to each ASIC via the four-wire IEEE 1149.1 interface, a Test Access Port (TAP) is included in the workstation libraries as a soft macro. TI's TAP (TS002), like all TAPs, is controlled by the TMS and TCK signals of the scan interface. The signals output by the TS002 are compatible with all the scan cells in the SCOPE library. Because the TS002 is a soft macro, any circuitry associated with an unused output can be removed.

To equip an ASIC with SCOPE, first complete the design of the chip's core application logic. Next, choose I/O buffers. The last step will be to add the SCOPE Cells to the design at each I/O pin along with the control circuitry necessary for IEEE 1149.1-compatible designs. The SCOPE circuitry should not interfere with the ASIC core logic design. Figure 1 illustrates the basic model for an ASIC design with SCOPE circuitry.

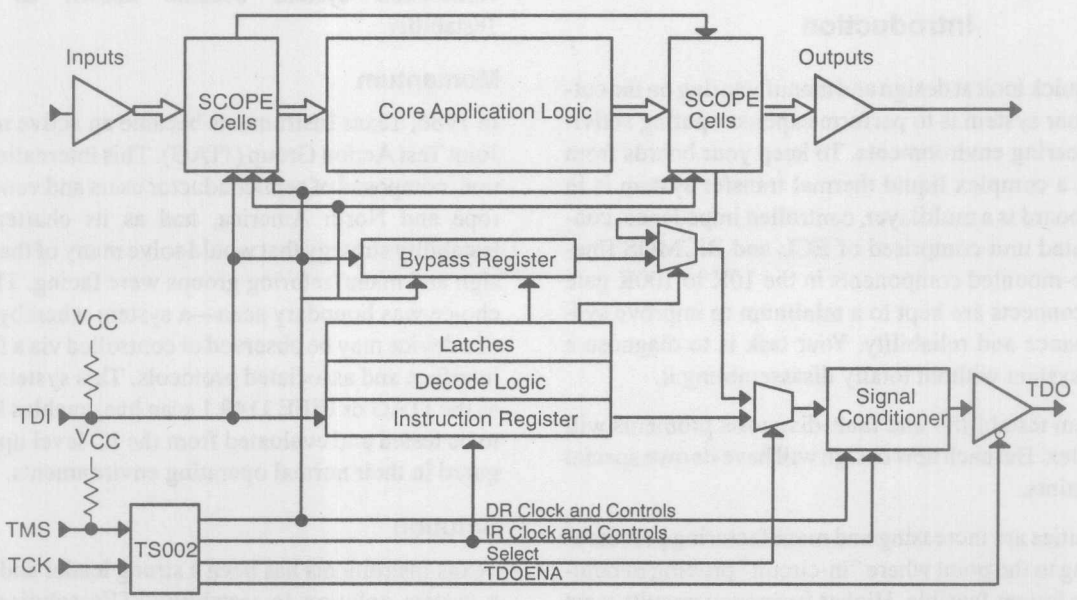


Figure 1. An ASIC with SCOPE

Several designs have been completed using the SCOPE libraries. Each works well and responds predictably to IEEE 1149.1 stimuli.

SCOPE Octal Test ICs: Many of today's designs still rely on standard octal bus interface devices for buffering signals between functional blocks of a system. Even in future designs, buffering signals at board edge connectors will continue to be a good idea to isolate complex devices from harmful off-board voltage spikes and to maintain backplane signal integrity. Typically, microprocessor and ASIC devices do not deliver the high drive necessary to push signals onto a backplane or into large blocks of RAM. Backplanes, microprocessors, and RAM have typically presented testability roadblocks to design engineers. With TI's SCOPE Octal Test ICs[6], a designer can incorporate boundary scan features (controllability/observability) into any or all octal devices on a board.

Currently, TI offers four SCOPE Octals: the 54/74BCT8244 octal buffer, 54/74BCT8245 octal transceiver, 54/74BCT8373 octal latch, and 54/74BCT8374 octal D flip-flop. All test features are accessed from the IEEE 1149.1 scan interface on each IC. In addition to controlling outputs and observing inputs of these devices, a pseudo-random pattern of data can be generated from the device outputs and a parallel signature analysis of the inputs can be made at the frequency of the Test Clock (TCK).

Future products are under consideration, but their availability will be driven by market demand and acceptance of the IEEE 1149.1 specification in general. Among the candidates for such new devices may include inverting bus functions, latches and flip-flops with alternate control inputs, and scannable field-programmable logic. Also being considered are bus monitoring devices that would enable observation of digital signals via the IEEE 1149.1 four-wire test bus.

SCOPE Test Bus Controller: To control the scan bus, a bus master IC has been developed. The SCOPE Test Bus Controller (TBC or 74ACT8990) receives commands and data from its host, and delivers IEEE 1149.1 control signals to the scan bus along with serial data. This device offloads the host from having to understand the scan protocol and speeds scan throughput. The TBC will be offered as part of TI's SCOPE family of components as an integral part of the testability and verification system. The TBC function may also be included in a future release of the ASIC workstation libraries as a soft macro or megacell.

Scan Controller Module: The TBC has been placed on the Scan Controller Module (SCM) card that fits inside an IBM PC/XT/AT compatible. The SCM is a half-length card composed of clock generation circuitry, discrete I/O circuitry, and the TBC device. With a cable connected between the SCM and the target system, fault diagnosis, system debug, and manufacturing verification can proceed.

High Speed Pod: Using TI's SCM, high speed scan operations can be implemented. Test clock speeds as high as 20 MHz have been tested with the TBC scanning to a demonstration system composed of four boards using prototype SCOPE Octal Test ICs. Depending on the system configuration, one or many high speed pods may be used to increase the distance of the target system from the test host.

Microprocessor Emulation: Texas Instruments TMS320C30 Digital Signal Processor[7] has built-in emulation functions accessible via a scan path. Plans for other application-specific microprocessors to incorporate emulation features accessible via an IEEE 1149.1 scan interface are in place. The advantages of this technology to the system, software, and design engineers are numerous. Specifically, the same four pins on a board edge connector used for testability purposes can be used for emulation purposes. One no longer has to remove the processor and insert a bulky emulator pod in its place for software development and system debug. All peripheral elements accessible by the processor are available through the emulator because the processor and emulator are one in the same. The customer no longer has to buy a separate emulator for each software development station. And, the emulator's speed is always that of the processor; no wait states need to be inserted.

ASSET Software: Tying the SCOPE hardware together is a software package tailored to serial scan test and debug strategies. The Advanced Support System for Emulation and Test (ASSET)[8] software development package was designed to help the engineer control test sequences through the scannable modules in the system without the tedium of handling scan protocols and keeping track of a dynamically changing scan path length.

The ASSET language is based on C++ functions and attributes but contains library functions tailored to scan test and debug scenarios. Included in the ASSET function libraries is a means to collect and view nodes sampled by the SCOPE Octals and scannable ASICs in a display format similar to a logic analyzer user interface. ASSET can be programmed to understand the architecture and testability features of an ASIC. It can take functional test patterns developed for an ASIC and scan them into the device to test it while the device is in the system. With the proper test program, faults from manufacturing or from environmental stress can be diagnosed.

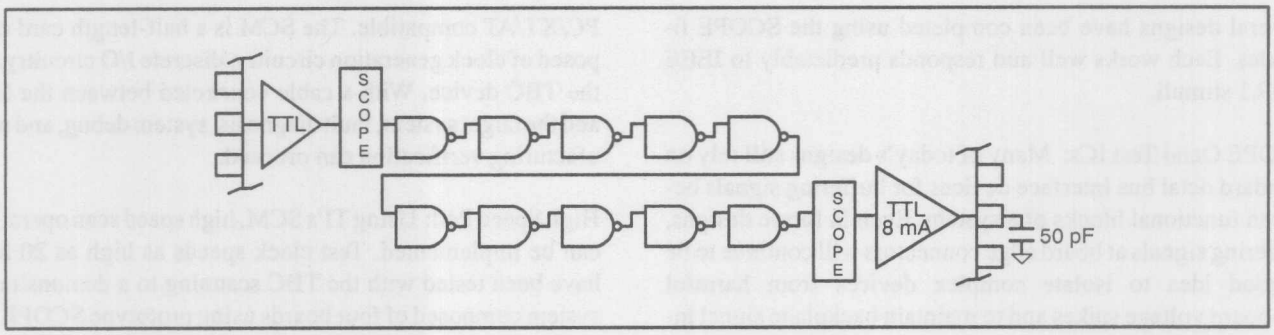


Figure 2. Delay Path with SCOPE Cells

ASSET is a modular and hierarchical package, and its routines can be called from other software packages. An interesting application could be written to extract data from design and manufacturing data bases of a particular system, and generate a test for all components in that system, as well as interconnections between components.

The ASIC Application

After completing several designs, some useful statistics can be shown.

Gate Overhead

To roughly calculate the number of extra gates necessary to add the SCOPE architecture to a 1-μm Standard Cell design using TI's TSC500 library components,[9] use the following formula:

$$G = IO \times 14.5 + IB \times 11 + TAP + N$$

where G = # gates to add
IO = # unidirectional I/Os
IB = # instruction register bits
TAP = # gates in TS002 Cell
N = variable from 50 to 300

This scenario assumes that an IEEE 1149.1-compatible design is to be achieved. Texas Instruments has standardized the Instruction Register length to be 8 bits. For IEEE 1149.1-only designs, only 2 bits are required. The TAP with all its outputs is 183 gates. The gate count drops to 154 after removing some specialized outputs. To complete the SCOPE architecture, an instruction decode circuit with latched outputs needs to be added along with TDO (Test Data Output) signal muxing and conditioning circuitry. The variable N in the equation accounts for this circuitry.

Speed Penalty

Adding the SCOPE circuitry to an ASIC design also exacts a speed penalty. Consider a SCOPE Cell placed between an input buffer and a 2-input NAND gate, or between a 2-input NAND gate and an output buffer: for all environmental conditions, one can say that in general, a 40 percent degradation in speed will be exacted on that net due to the inclusion of a 2:1 mux in the data path. For example, if the delay associated with a net driven by an input buffer to a 2-input NAND gate is 1.6 nanoseconds (typical for TI's 1-μm Standard Cell), then, after adding a SCOPE Cell to the net, the delay between the input buffer and NAND gate would typically be about 2.3 nanoseconds.

To further illustrate the small speed degradation of a device using the SCOPE architecture, let us assume an ASIC device with SCOPE cells is designed as illustrated in Figure 2.

Without the SCOPE Cells, the typical device propagation delay would be 8.42 nanoseconds. Adding the SCOPE Cells at the input and output buffers increases the typical propagation delay to 10.09 nanoseconds. Adding the SCOPE Cells to the boundary added only 1.67 nanoseconds to the typical device propagation delay. The typical propagation delay through a SCOPE Cell is 0.7 nanoseconds.

Internal Scan

Partitioning: Because the SCOPE Cell component is transparent in nature when the device is in a normal functional mode, SCOPE Cells can be used internal to the ASIC to control and observe internal nodes of the circuit. The IEEE 1149.1 architecture allows for as many user data paths as a designer sees fit to include in the design. These internal paths may be used to surround and isolate an internal function in much the same manner as boundary scan is used to isolate the ASIC itself from the rest of the system for testing, fault diagnostic, and debugging purposes.

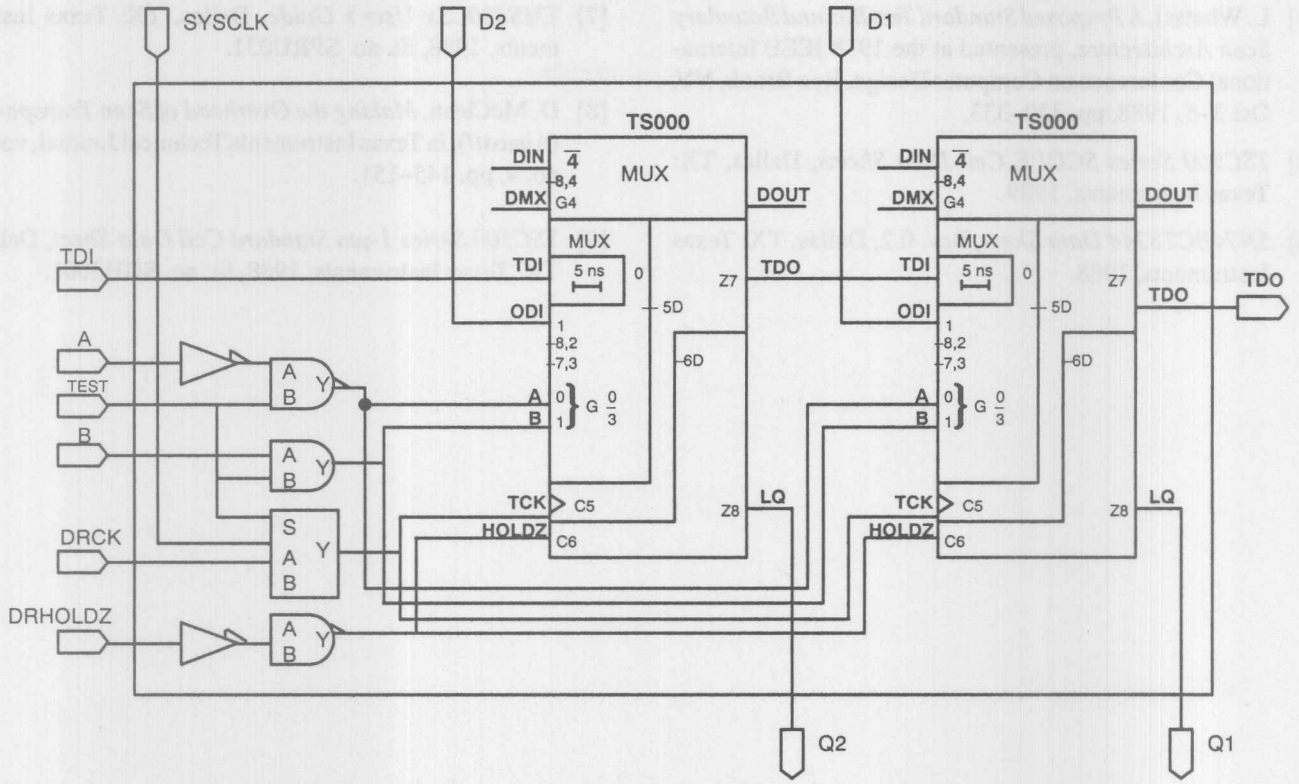


Figure 3. D-Flip-Flops

Scannable Register: The SCOPE Base Cell (TS000) may be used as a scannable flip-flop. This is a useful application of the SCOPE Cell to gain the controllability/observability functions in the core-application sequential logic. Figure 3 shows a typical application of this methodology.

The TEST signal comes from the test instruction decode logic, or an IEEE 1149.1 User Data Register. It controls the mode of the flip-flops. Signals on each side of the D-Flip-Flops box come from SCOPE test logic circuitry. Signals above and below the box represent normal function inputs and outputs.

Conclusions

The SCOPE boundary scan architecture can be used in conjunction with the post-packaging test vector set to perform a complete, high-confidence test of the ASIC. By also incorporating internal scan-between functional circuit blocks, higher controllability/observability partitions can result in better fault coverage, smaller vector sets, and shorter test times. In addition, these scan paths may be used to emulate peripheral functions to aid in the system debugging and verification processes.

After SCOPEing a system, each scannable node can become a virtual logic analyzer probe or pattern generator clip. With ASSET software controlling your system, the design, manufacturing, test, field service, and software engineers can leverage their own testing, diagnostic, and development needs from the same circuitry–all without physically touching the system.

Future developments within Texas Instruments and other industry leaders should produce new and valuable additions to the IEEE 1149.1-compatible product lines. As interest and knowledge increases, integration of this technology will become a matter of course for IC, board, and system vendors.

References

[1] *Testability Test and Emulation Primer,* Dallas, Tx: Texas Instruments, 1989, lit. no. SSYA002.

[2] *A Test Access Port and Boundary Scan Architecture*, Ver. 2.0, Technical Sub-Committee of the Joint Test Action Group, 1988.

[3] L. Whetsel, *A View of the JTAG Port and Architecture*, Proceedings of the ATE and Instrumentation Conference, 1988, pp. 385–401.

[4] L. Whetsel, *A Proposed Standard Test Bus and Boundary Scan Architecture*, presented at the 1988 IEEE International Conference on Computer Design, Rye Brook, NY, Oct 3–5, 1988, pp. 330–333.

[5] *TSC500 Series SCOPE Cell Data Sheets*, Dallas, TX: Texas Instruments, 1989.

[6] *SN74BCT8244 Data Sheet*, Rev. 0.2, Dallas, TX: Texas Instruments, 1988.

[7] *TMS320C3x User's Guide*, Dallas, TX: Texas Instruments, 1988, lit. no. SPRU031.

[8] D. McClean, *Making the Overhead of Scan Transparent (Almost!)*, in Texas Instruments Technical Journal, vol. 5, no. 4, pp. 145–151.

[9] *TSC500 Series 1-μm Standard Cell Data Sheet*, Dallas, TX: Texas Instruments, 1988, lit. no. SGH3001.

Impact of JTAG/1149.1 Testability on Reliability

by Alfred L. Crouch and Carol Pyron

This paper was presented at the Government Microcircuit Applications Conference (GOMAC), 1989.

Introduction

Military programs are imposing increasingly aggressive testability requirements on modern electronics development programs. These requirements usually increase the size of the hardware design. Traditionally, the addition of hardware for testability functions such as readback, pattern generation, pattern insertion, and functional hardware test control is known as ad hoc testability. In general, testability hardware can increase design factors such as part count, connector pins, board "real estate," cost, and power. Various Department of Defense (DoD) requirements, such as MIL-STD-2165[1] or DARCOMP 34–1[2], provide guidance to the application of ad hoc testability.

JTAG/1149.1 Overview

Recent advances in testability methodology have led to the development of bus-based observability and controllability systems instead of the traditional ad hoc methods. One of these bus-based systems was proposed by the Joint Test Action Group (JTAG). The specification developed by JTAG has evolved into the IEEE 1149.1 standard.[3] This standard is a four-wire test bus interface and boundary scan architecture specification. Boundary scan is a specialized scan path that provides observability and controllability to device or board input/output pins. 1149.1 is a testability architecture designed to improve a circuit's fault detection and isolation capabilities. Products such as test octals have been developed to incorporate the boundary scan protocols.

Testability Design Tradeoffs

Testability can impact related design support disciplines such as reliability, maintainability, or producibility. The addition of testability circuitry can reduce test costs and time, improving the ease and accuracy of fault detection and isolation. Testability can improve maintainability calculations for Mean Time To Repair (MTTR) by decreasing the test isolation time but can impact the reliability calculation for Mean Time Between Failure (MTBF) negatively by increasing the failure rate.

The ad hoc testability approach typically adds 10 to 15 percent extra hardware to the functional design. The Texas Instruments approach to the application of 1149.1 is to place testability features into already required functional parts.

Two of the new 1149.1-based products are used in a Texas Instruments system design. The products currently offered are specialized test octal buffers, latches, transceivers, and registers with four extra pins and circuitry for support of the 1149.1 standard. The parts are from the Texas Instruments SCOPE family. These test octals exceed the 1149.1 requirements by also supporting additional testability functionality such as Parallel Signature Analysis (PSA) and Pseudo-Random Pattern Generation (PRPG) (Table 1).

Individually, a SCOPE test octal has approximately 800 gates, as compared to less than 100 in a standard octal. The majority of the additional gate count in the scan parts is not in the functional path but in the scan path control logic. The addition of boundary scan to a buffer places only two gates in each functional signal path. Only 16 gates are added to the normal operational mode. A failure in the scan path logic does not affect the chip's normal function in most cases. A failure in the scan path control and scan logic can be isolated by using the inherent parallelism of the boundary scan logic. The extra gate count impacts the chip-level failure rate, but the benefits for board- and system-level testing can justify the increased failure rate.

Table 1. Feature Comparison for Standard Octal and SCOPE Testability Octal Parts

Features	Standard Octal Parts	Testability Octal Parts
Pin count	20	24
Gate count	< 100	~800
Failure rate:		
at 0°C	0.0401	0.0563
at 60°C	0.1500	0.2762
Normal functions:	Buffer, latch, transceiver, register	Buffer, latch, transceiver, register
Internal test functions for testing of other parts	None	Signature analysis Pseudorandom pattern generation Boundary scan Readback and latch (245) 1/0 toggle mode
External test purposes	Readback latch control register	Readback latch Control register Pattern generator

Memory Board Example

This paper illustrates the impact of 1149.1 on reliability with a basic memory board example. The memory board example was selected because it is a common design architecture, and because it requires additional testability for fault detection and isolation. To fully illustrate the design tradeoffs involved, the memory board is implemented in the following three different configurations:

- No testability baseline design (Figure 1)
- Ad hoc testability design (Figure 2)
- 1149.1 testability design (Figure 3)

Each of these three designs has different tradeoffs for reliability, testability, and other design considerations.

Baseline Memory Board Design

The baseline memory board is a straightforward simple design consisting of 17 parts on a Versabus Modular European (VME) type board (Figure 1 and Table 2). The data and address buses are each 16-bits wide, and the processor control bus is 8-bits wide. These buses are pulled-up and buffered between the memory and the connector. The decode function consists of a comparator for the board base address select and a Programmable Array Logic (PAL) to implement memory select, address decode, and read/write control. The board's base address is jumpered and buffered. Some additional control logic (such as write-protect enable) is implemented with combinational gates.

The baseline memory board is not inherently testable for isolation purposes. To meet today's requirements, component ambiguity groupings of four or less must have isolation percentages in the high 90s. Despite the simplicity of the board, it cannot meet these requirements.

For an example of the poor fault isolation capability, examine the following case. A failure is detected by a test sequence that writes to the Static Random-Access Memory (SRAM) and reads back the written value for comparison. The following are possible candidates for the fault location if the test fails.

- A stuck-at bit in the SRAM—returns the wrong value on the read
- The bidirectional transceiver—disrupts the data bus transfer
- The decode logic—chooses the wrong memory device or wrong peripheral device (the I.D. buffer)
- The address decode—chooses the wrong memory location
- The control logic—chooses wrong memory action (read versus write)
- Identification jumper/buffer—chooses the wrong board in the system or maps memory incorrectly
- The connector—passes incorrect data
- The resistor pack—causes a bus line to float.

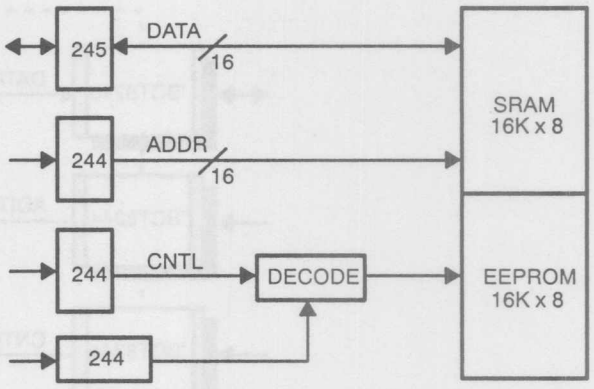


Figure 1. No Testability Baseline Design

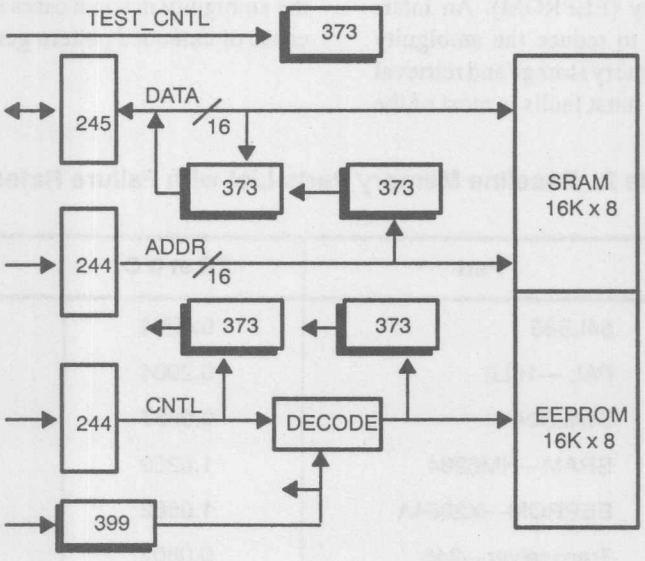


Figure 2. Ad Hoc Testability Design

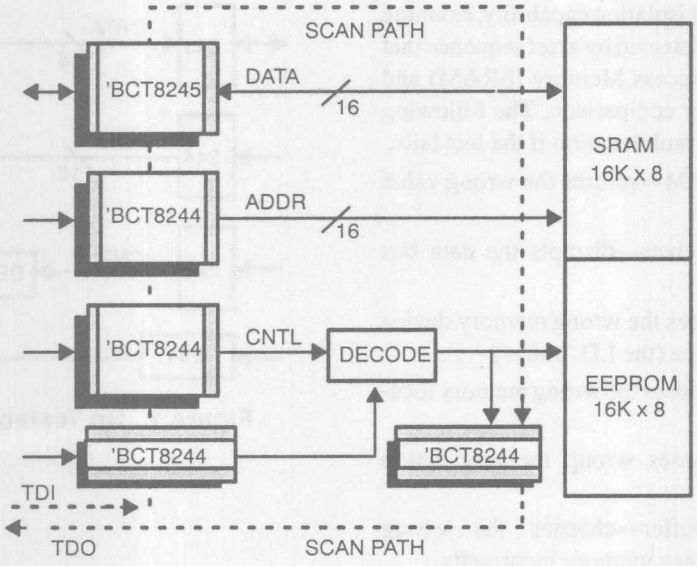


Figure 3. 1149.1 Testability Design

Any of these failures can cause the return of an incorrect data value. Only two types of tests can be run on this board: an end-to-end test of the SRAM or of the Electrically Erasable Programmable Read-Only Memory (EEPROM). An intermediate test cannot be performed to reduce the ambiguity groupings. All testing requires a memory storage and retrieval to detect a fault. The ambiguity for most faults is most of the board (Table 3).

To reduce the ambiguity requires long, complicated testing algorithms that run specific pattern sequences through the memory space to characterize the fault. This only can reduce the ambiguity in some cases and results in long test times because of extended pattern generation.

Table 2. Baseline Memory Parts List with Failure Rates

Qty	Part	FR at 0°C	FR at 60°C
1	54LS85	0.0294	0.0997
1	PAL —16L8	0.2904	1.6314
1	54ALS04	0.0237	0.0519
2	SRAM—HM6264	1.6259	23.6086
2	EEPROM—X2864A	1.0582	9.4955
2	Transceiver—245	0.0802	0.3000
4	Buffer—244	0.1604	0.6000
1	Jumper	0.0000	0.0000
2	Resistor pack	0.0807	0.7631
2	96-pin connector	0.0646	0.2908
	Failure rate totals =	3.4135	36.8410

Table 3. Baseline Memory Board Test Flow and Fault Isolation Ambiguity Groupings

Baseline Test Flow	Ambiguity Groupings
SRAM tests	17—Connector, Pullups, buffer, comparator, PAL, SRAM, EEPROM, I.D. buffer, jumpers, transceiver, NANDs
EEPROM tests	17—Connector, Pullups, buffer, comparator, PAL, SRAM, EEPROM, I.D. buffer, jumpers, transceiver, NANDs

Table 4. Ad Hoc Memory Board Parts List with Failure Rates

Qty	Part	FR at O°C	FR at O°C
1	54LS85	0.0294	0.0997
1	PAL—16L8	0.2904	1.6314
1	54ALS04	0.0237	0.0519
2	SRAM—HM6264	1.6259	23.6086
2	EEPROM—X2864A	1.0582	9.4955
2	Transceiver—245	0.0802	0.3000
*3	Buffer—244	0.1604	0.6000
1	Jumper	0.0000	0.0000
2	Resistor pack	0.0807	0.7631
*1	Multiplexer latch	0.0564	0.1267
*7	Octal D-register	0.2807	1.0500
2	96-pin connector	0.0646	0.2908
*	>60 extra pins used		
	Failure rate totals =	3.7105	37.8677

* Differences from the baseline memory board

Ad Hoc Testability Memory Board

The ad hoc version of the memory board is derived from the baseline board by placing readback latches on the data, address, and control buses, by replacing the ID buffer with a 4-bit multiplexer/latch; and by placing a test control register on the board (Figure 2, Table 4). The design contains the minimum testability required to bring the ambiguity groupings to four components or less. The testability improvement results in adding seven parts to the board and replacing another.

The ad hoc testability simplifies the isolation of a detected fault. Table 5 identifies a specific test sequence that will result in the minimum ambiguity groupings.

The first test should be to write a base address to the I.D. latch through the added multiplexer. This allows the tester to test all memory boards identically, regardless of the jumpered ad-

dress. Next, all buses should be written to and read back by the '373 latches. This intermediate test allows faults on the connectors, buffers, and transceiver to be isolated. The next test should ensure that the decode function selects the correct memory and addresses. Finally, the memory is tested.

By partitioning the tests as shown in Table 5, the fault can be isolated by a divide-and-conquer approach. The benefits are fewer patterns, faster test time, and a higher confidence of isolating a fault. The cost is seven added parts and the related design penalties.

1149.1 Testability Memory Board

In this case, scannable buffers and transceivers replaced the original buffers and transceivers (Figure 3, Table 6). One additional part provides observability and controllability on the memory side of the decode block. Then, the SCOPE octal parts were connected into a single scan path.

The 1149.1 board is more inherently testable than the ad hoc board since the boundary scan allows the insertion and overwrite of signals on the device output pins. The I.D. jumpers can be reconfigured without using a discrete multiplexer. The scan path allows patterns to be read back between the buffer and the connector. This removes the connector from the ambiguity group. By inserting the patterns through the scan path and following the same test sequence as the ad hoc board, the isolation percentages are improved. No ambiguity group has more than two components (Table 7).

An additional advantage of the scan path method is that the memories do not have to be tested last in the sequence. The generated patterns and control of the memory can be done via the scan path directly, without having to use the decode logic. This allows isolation without the restriction of a specific test sequence. The test sequence can be redesigned to a failure rate priority that would have a probability of finding a fault earlier in the test sequence.

The overall advantages of using the SCOPE Octal parts versus an ad hoc solution is a reduction in board space used and the ability to modify the test sequence.

Analysis of Results

The failure rate analysis was conducted as per MIL-HDBK-217E.[4] Each of the three boards described above was analyzed using a ground mobile model at 0°C ambient (with an equivalent junction temperature of 50°C) and at 60°C ambient (with an equivalent junction temperature of 110°C). The devices were modeled as full MIL-qualified parts. To keep the comparison similar, all the added parts involved were Bipolar Complementary Metal-Oxide Semiconductor (BiCMOS) process, since the SCOPE Octal parts are only available in that technology.

The overall reliability comparison figures for the three boards are shown in Table 8.

The percentage of test circuitry added to the baseline board is shown in Table 9.

Conclusions

The memory board examples analyzed above are a single, small test case; therefore, the testability modifications have a greater impact on reliability than when used on other, more realistic designs.

As seen in Table 8, the ad hoc solution has a higher failure rate at low temperature where the predominant failure mode is the interconnect. The 1149.1 solution has a slightly higher failure rate at the high temperature where the predominant failure mode is the silicon process. These results are expected since the ad hoc solution adds parts to the board and the 1149.1 solution adds gates to the parts.

Table 5. Ad Hoc Memory Board Test Flow and Fault Isolation Ambiguity Groupings

Ad Hoc Test Flow	Ambiguity Groupings
I.D. overwrite	1—Latched Multiplexer
Pattern data	4—Connector, buffer, latch, pullup
Pattern address	4—Connector, latch, transceiver, pullup
Pattern control	4—Connector, buffer, latch, pullup
Pattern decode	2—Comparator, PAL
SRAM tests	3—SRAMs, NANDs
EEPROM tests	3—EEPROMs, NANDs

Table 6. 1149.1 Memory Board Parts List with Failure Rates

Qty	Part	FR at O°C	FR at 6O°C
1	54LS85	0.0294	0.0997
1	PAL—16L8	0.2904	1.6314
1	54ALS04	0.0237	0.0519
2	SRAM—HM6264	1.6259	23.6086
2	PROM—X2864A	1.0582	9.4955
*2	Test transceiver—'BCT8245	0.1126	0.2762
*5	Test buffer—'BCT8244	0.2815	1.3810
1	Jumper	0.0000	0.0000
2	Resistor pack	0.0807	0.7631
2	96-pin connector	0.0646	0.2908
*	Four extra pins used		
	Failure rate totals =	3.5670	37.8744

* Differences from the baseline memory board

Table 7. 1149.1 Memory Board Test Flow and Fault Isolation Ambiguity Groupings

1149.1 Test Flow	Ambiguity Groupings
Scan path	1—Each test part
I.D. overwrite	1—I.D. register
Pattern connector	1—Connector
Pattern data	2—Test buffer, pullup
Pattern address	2—Test transceiver, pullup
Pattern control	2—Test buffer, pullup
Pattern decode	2—Comparator, PAL
SRAM tests	3—SRAMs, NANDs
EEPROM tests	3—EEPROMs, NANDs

From a reliability point of view, the ad hoc and 1149.1 solutions have similar impacts on board failure rate (Table 9). For this example, 1149.1 has an improved overall numerical impact. In terms of percentage of failure rate added to the board, the 1149.1 actually has the least maximum impact with 4.45 percent additional failure rate versus the ad hoc maximum of 8.70 percent.

The 1149.1 board surpassed the isolation requirement of four components with no ambiguity group larger than two (Table 7), whereas the ad hoc board just met the requirement (Table 5).

If the ad hoc board were implemented with all of the testability functionality contained within the test octals, the part count would increase significantly. The ad hoc board failure rate would then surpass the 1149.1 solution significantly.

Another advantage the 1149.1 solution has over the ad hoc solution is the number of test points required. The 1149.1 standard requires only four connector pins to be added. In this case, the ad hoc solution required in excess of 60 test points at the connector.

Table 8. Reliability Comparisons

Reliability Comparison	Baseline Board	Ad Hoc Board	1149.1 Board
Failure rate			
at 0.0°C	3.4135	3.7105	3.5670
at 60.0°C	36.8410	37.8677	37.8744
Mission failure	20.1273	20.7891	20.7207
(FM) f/mh			
MTBF hours	49,700	48,100	48,300

Table 9. Percentage Added Because of Testability Circuit

Impact of Testability	Ad Hoc Board (%)	1149.1 Board (%)
Part count	41.17	5.88
Failure rate		
at 0.0°C	8.70	4.45
at 60.0°C	2.79	2.80

1149.1 testability solutions, as compared to ad hoc testability solutions, offer the following advantages.

- Improved observability and controllability
- Improved fault detection and isolation
- Improved reliability
- Fewer test points
- Reduced part count
- Reduced pattern sets
- More flexible test flow structure
- Lower power consumption
- Better thermal profile

Acknowledgments

The authors recognize the contributions and assistance provided by Bill Grimes and Bill Waite of Texas Instruments.

References

[1] MIL-STD-2165. *Testability Program for Electronic Systems and Equipments.*

[2] DARCOMP 34–1. *Built-In Test Design Guide.*

[3] *IEEE P1149.1 Draft 6 Standard Proposal.*

[4] MIL-HDBK-217E. *Reliability Prediction for Electronic Equipment.*

JTAG-Compatible Devices Simplify Board-Level Design for Testability

by Lee Whetsel

This paper was presented at the WESTCON Conference, 1989.

Introduction

A specification for an integrated circuit (IC) test bus and boundary scan architecture has been developed by the Joint Test Action Group (JTAG).[1,2] This specification has been endorsed by the IEEE 1149 Standards Committee. This testability standard, referred to as IEEE 1149.1, promises to significantly improve IC and board level testing.[3,4,5] However, before the industry truly can evaluate the benefits of 1149.1, hardware components supporting the standard must be made available.

Texas Instruments (TI), a participant in the development of the JTAG standard specification, has taken an active role in producing 1149.1-compatible devices required to implement the standard. These devices are members of TI's System Controllability, Observability, and Partitioning Environment (SCOPE) family of testability products. While all SCOPE products conform to 1149.1, some include extensions to improve the ability to perform scan and test operations further at the IC through system levels. The SCOPE products previewed in this paper include an 1149.1 Test Bus Controller (TBC) IC, scan path support IC, a series of SCOPE octal ICs, and a Digital Bus Monitor (DBM) IC. While products presented in this paper are described in detail, the exact implementation and/or operation are subject to change.

Elements of an 1149.1-Based Design

When implementing 1149.1 into a product, designers must have access to a range of test hardware building blocks. Application-Specific Integrated Circuit (ASIC) designers need a library of test cells to implement 1149.1 efficiently at the IC level. Similarly, board and system designers need a variety of off-the-shelf test components to implement 1149.1 at higher levels of assembly. While SCOPE provides both ASIC cells and standard components supporting 1149.1, the latter is the subject of this paper.[6,7] The SCOPE products presented in this paper fall under two categories: products supporting the design of system scan path architectures (TBC and scan path support ICs), and products supporting board-level design for testability (test octals and DBM ICs).

SCOPE Test Bus Controller (TBC) IC

One of the key features provided by the 1149.1 standard is a common IC-level serial testability bus. This four-wire test bus forms the common thread through which different merchant ICs may be linked together at the board level to effect testing. With the realization of a standard IC-level test bus, a need has arisen to develop test bus controller ICs to support efficient transfer of serial data and control to and from target devices on the serial test bus.[8]

Figure 1 shows an example application of the TBC. The TBC provides the hardware link between a host processor and target devices on the serial scan path. To the processor, the TBC is a peripheral mapped into a particular area of the processor's external memory space. The TBC operates to transfer data and control between the parallel processor and target devices on the serial test bus. The TBC's processor interface consists of a 16-bit-wide bidirectional data bus; inputs for address, read/write, and chip select signals; and an interrupt output signal.

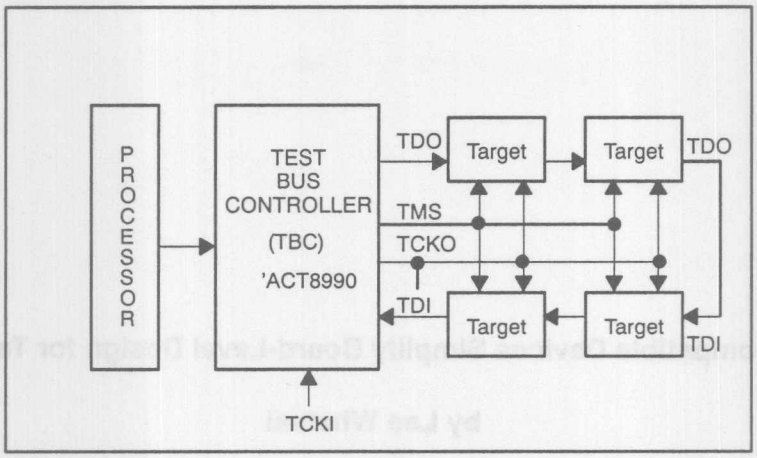


Figure 1. Test Bus Controller Example

The TBC's test bus interface consists of a Test Data Output (TDO), Test Data Input (TDI), Test Mode Select (TMS) outputs, and a Test Clock Input (TCKI), and Test Clock Output (TCKO). The TMS outputs allow the TBC to support separate scan paths. TDO is a serial-data output from the TBC that drives the TDI input of the first target device in the scan path. TDI is a serial-data input to the TBC that receives the TDO output from the last target device in the scan path. The TMS signals are serial control outputs from the TBC that drive TMS inputs of target devices in the scan path. The TMS output from the TBC conforms to the protocol described in the 1149.1 standard to cause target devices on the scan path to shift data from their TDI input to their TDO output. The TCKI signal is generated externally and is input to the TBC, and is distributed via TCKO to each target device on the scan path. In addition to the required 1149.1 test bus signals, the TBC interface includes an output signal for initialization of target devices and input signals for receiving test-related interrupts from target devices.

Before a scan operation, the TBC receives the parallel data input from the processor that is to be transmitted serially to the target devices in the scan path. Also, the processor inputs a count value into an internal TBC counter, specifying the number of serial data bits to be transferred. After the data and count values have been set up, the TBC receives a command from the processor to initiate the scan operation. During scan operations, the TBC outputs serial data and control signals to the target devices via the TDO and TMS output signals and receives serial data from the target devices via the TDI input signal. By reading status bits from the TBC, the processor determines when the TBC requires additional read and write operations to maintain the flow of serial data to the target devices in the scan path. When the TBC's internal counter reaches a minimum value, the TBC outputs an interrupt to the processor, indicating that the required number of serial data bits has been shifted through the scan path.

In addition to controlling scan operations, the TBC simplifies the execution of Built-In Self-Test (BIST) features incorporated in the target device shown in Figure 1. The 1149.1 test bus protocol state diagram includes a steady state, referred to as run test/idle, in which BIST operations may be executed. If a target device includes a BIST capability, an instruction invoking the BIST operation can be scanned into the device. After the target device receives the BIST instruction, execution of the test occurs when the TBC transitions the test bus into the run test/idle state. As described in the 1149.1 specification, the length of a BIST operation is defined by a number of TCK inputs applied while the test bus is in the run test/idle state.

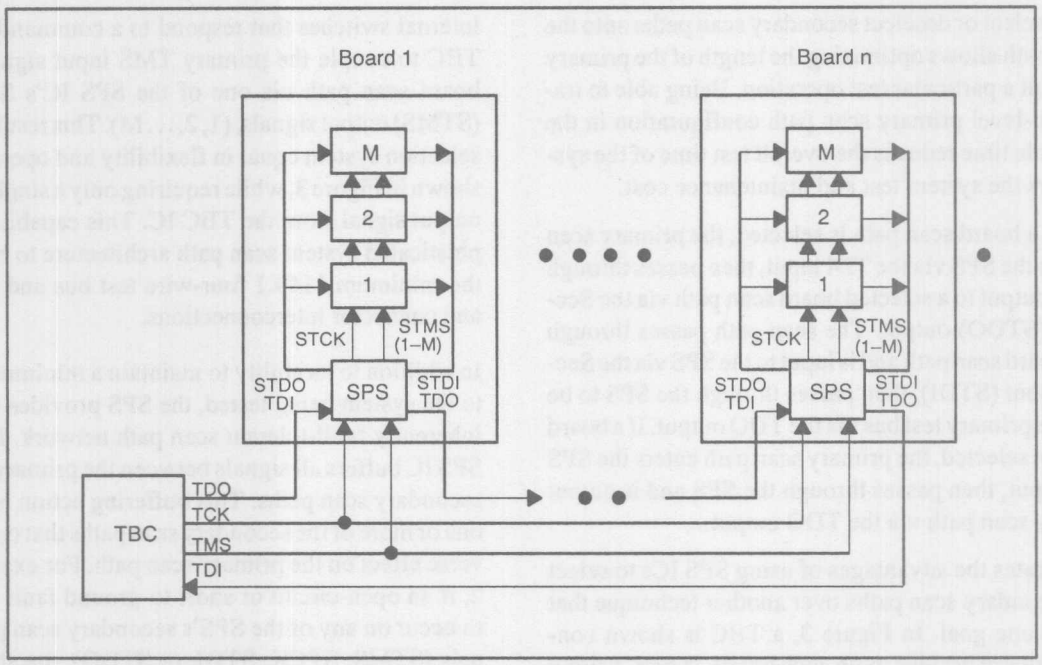


Figure 2. Scan Path Selector Example

To simplify the execution of 1149.1 BIST operations, the TBC contains two types of run test commands. Both command types are executed while the test bus is in the run test/idle state. The first type of run test command uses the TBC's internal counter to count the number of TCKs applied while the test bus is in the run test/idle state, supporting the BIST procedure detailed in the 1149.1 specification. When the counter reaches a minimum value, the TBC transitions the test bus from the run test/idle state into either a scan or pause state to terminate the execution of the BIST operation. The second type of run test command uses the TBC's interrupt inputs to determine the length of time the test bus is in the run test/idle state. This command differs from the first in that it assumes the target devices have additional test pins from which an end-of-test interrupt may be issued to the TBC. When the TBC receives an end-of-test interrupt, the TBC transitions the test bus from the run test/idle state to either a scan or pause state to terminate the execution of the BIST operation.

TBC ICs will be available in 44-pin PLCC (Plastic Leadless Chip Carrier). The TBC supports a maximum scan rate of 30 MHz. At higher scan rates, the TBC may require pausing the test bus for the processor to load and unload data. In addition to its 1149.1 capabilities, the TBC also supports the test interface required for in-circuit emulation of TI's microprocessor ICs.

SCOPE Scan Path Selector (SPS) IC

The SPS IC is a first in the series of devices that simplify the design of system-level scan path architectures. The SPS IC provides programmable scan access to multiple boards in a system design. It selectively routes the test bus through different scan path channels without requiring the use of multiple TMS control signals.

As shown in Figure 2, the SPS IC provides a method of coupling and decoupling board-level secondary scan paths to a primary scan path. The primary scan path is defined as the system scan path directly coupled to the TBC IC or other test bus-controlling source. The secondary scan path is defined as one or more selectable board-level scan paths directly coupled to the SPS IC. In some instances, an SPS may be considered a test-controlling source if it issues control to another SPS residing in one of its selectable secondary scan paths. In this case, the scan path output from the controlling SPS is considered a primary scan path, and the scan path output of the SPS being controlled is considered a secondary scan path. This scenario can be replicated to create a vertical, as well as a horizontal, hierarchy of primary and secondary scan path networks throughout a system scan path architecture.

As shown in Figure 2, SPS ICs existing on a primary scan path can be set by the TBC to select one of the board-level secondary scan paths (1-M), or bypass the board scan paths entirely.

The ability to select or deselect secondary scan paths onto the primary scan path allows optimizing the length of the primary scan path to suit a particular test operation. Being able to traverse a system-level primary scan path configuration in the shortest possible time reduces the overall test time of the system and lowers the system test and maintenance cost.

In Figure 2, if a board scan path is selected, the primary scan path is input to the SPS via the TDI input, then passes through the SPS to be output to a selected board scan path via the Secondary TDO (STDO) output. The scan path passes through the selected board scan path and is input to the SPS via the Secondary TDI input (STDI), then passes through the SPS to be output onto the primary test bus via the TDO output. If a board scan path is not selected, the primary scan path enters the SPS via the TDI input, then passes through the SPS and is output on the primary scan path via the TDO output.

Figure 3 illustrates the advantages of using SPS ICs to select or deselect secondary scan paths over another technique that achieves the same goal. In Figure 3, a TBC is shown connected to "n" board designs, with each board, in turn, having "M" selectable scan paths. To gain access to one of the scan paths (1, 2, . . . M) in each board design (1, 2, . . . n), the TBC must have a number of TMS output signals equal to the sum of the total scan paths of each board in the system, i.e., the number of TMS output signals equals ($M_1 + M_2 + M_3 + \ldots M_n$). For example, if a system has 20 boards, with each board having five individually selectable scan paths, the total number of TMS output signals required from the TBC would take up 100 of the IC's package pins. To make matters worse, the number of interface signals to perform scan access to each board would require a test interface cable and connector bus width of 103 wires, 100 for the TMS signals and three for the TCK, TDI, and TDO signals.

By using the SPS IC to design the system scan path architecture, as shown in Figure 2, a TBC with only a single TMS output signal can gain individual access to each scan path (1, 2, . . . M) of each board design (1, 2, . . . n) in the system. This is possible because the SPS IC contains control circuitry and internal switches that respond to a command input from the TBC to couple the primary TMS input signal to a selected board scan path via one of the SPS IC's Secondary TMS (STMS) output signals, (1, 2, . . . M). This results in a scan path selection system equal in flexibility and operation to the one shown in Figure 3, while requiring only a single primary TMS output signal from the TBC IC. This capability allows a sophisticated system scan path architecture to be supported by the minimum 1149.1 four-wire test bus and related cabling and connector interconnections.

In addition to its ability to maintain a minimum test interface to the system being tested, the SPS provides the basis for an inherently fault-tolerant scan path network. In Figure 2, the SPS IC buffers all signals between the primary and selectable secondary scan paths. This buffering action isolates faults in one or more of the secondary scan paths that could have an adverse effect on the primary scan path. For example, in Figure 2, if an open-circuit or short-to-ground fault condition were to occur on any of the SPS's secondary scan path output signals (STMS, STCK, STDI, or STDO), the shifting of serial data into the secondary scan path would be inhibited. Faults of this nature could be tolerated until repaired by simply deselecting the faulty board from the primary scan path so that the primary scan path only passes through the SPS from the primary TDI input to the primary TDO output. In this way, a faulty section in a system scan path network can be bypassed effectively to maintain access to other scan paths in the system. In the example scan path select scheme shown in Figure 3, a board-level scan path short- or open-circuit fault would disable the operation of the entire system scan path network since no means of bypassing a board is available.

The SPS IC will be available in 28-pin DIP and SOIC, or 28-pin LCC. The SPS includes an 8-bit bidirectional data base that serves as both a board identification input and communications port between a board resident test processor and the TBC. A second device is the Scan Path Linker (SPL). The SPL is similar to the SPS, except that the SPL can select any of the four secondary scan paths together.

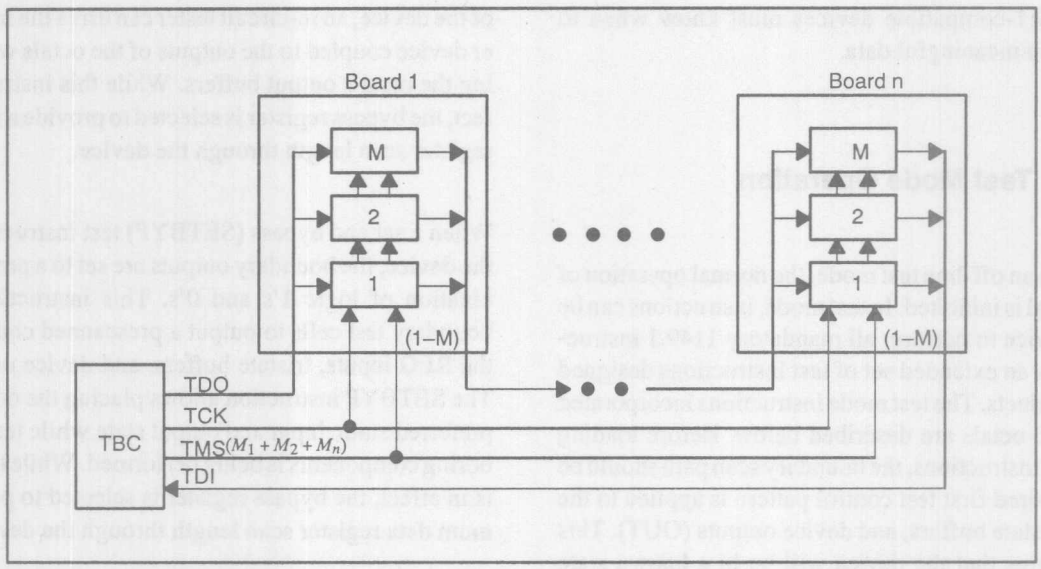


Figure 3. Alternate Scan Path Configuration Example

SCOPE Octal ICs

The SCOPE octals are the first in a series of standard components to be offered by TI that blend functionality with embedded board-level testing features.[9] The initial products include an octal register type 374, latch type 373, buffer type 244, and transceiver type 245. Along with the normal function pins associated with each part, four pins are added to support the 1149.1 test bus interface signals: TDI, TMS, TCK, and TDO. These devices can be substituted for their nontesting counterparts in a variety of board-level design applications such as pipeline registers, board I/O buffers, address and data buffers/transceivers, and finite state machine designs.[10]

An architectural illustration of the 374 octal register type is shown in Figure 4. The functional architecture of the 374 octal consists of an 8-bit register (REG), 8 data inputs (IN), 8 data outputs (OUT), a clock input (CK), and an tristate output control input (OC). The 1149.1 architecture consists of a Test Access Port (TAP) controller, an instruction register (IREG), and a data register section. The data register section consists of a bypass register, a Boundary Control Register (BCR), and a boundary scan register. The boundary scan register consists of SCOPE test cells 1 and 2 (TC1, TC2) coupled to the CK and OC inputs, SCOPE test cell register 1 (TCR1) coupled to the IN inputs, and SCOPE test cell register 2 (TCR2) coupled to the OUT outputs. The other SCOPE octals have a similar 1149.1 test architecture placed around buffer, transceiver, and latch functions.

The TAP controller receives external input from the TMS and TCK signals and outputs internal control to either the IREG or a selected data register, causing a shift operation from the TDI input to the TDO output. The IREG stores a test instruction to be executed by the IC. The bypass register shortens the scan path length through the IC to a single bit during data register operations. The BCR stores boundary configuration control bits to extend the test capabilities of the boundary scan register. The boundary scan register provides the mandatory test features required for 1149.1 compatibility, as well as extended test features developed for SCOPE products.

Normal Mode Operation

During normal operation, the boundary scan register is transparent, allowing input and output signals to pass freely through the test cells and enabling the device to perform its intended function. While in normal operation, the TAP can receive control from the TMS and TCK inputs to shift data through the device from the TDI input to the TDO output. Three test instructions can be executed while the device is in normal mode: 1149.1 sample and bypass instructions and a SCOPE cell self-test instruction. The sample instruction allows the data flowing through the boundary to be sampled and then shifted out for inspection. The bypass instruction selects the bypass register to be shifted during data register scan operations, reducing the scan path length through the device to 1 bit. The self-test instruction executes a self-check of each SCOPE cell in the boundary scan register. While the sample instruction at first may appear very attractive, the user of these

and other 1149.1-compatible devices must know when to sample to obtain meaningful data.

Test Mode Operation

When placed in an off-line test mode, the normal operation of the SCOPE octal is inhibited. In test mode, instructions can be input to the device to perform all mandatory 1149.1 instructions, as well as an extended set of test instructions designed for SCOPE products. The test mode instructions incorporated into all SCOPE octals are described below. Before loading these test mode instructions, the boundary scan path should be set so that a desired first test control pattern is applied to the REG inputs, tristate buffers, and device outputs (OUT). This procedure ensures that the device will be in a known state when the test mode is entered.

When either an external or internal 1149.1 boundary scan test (EXTEST or INTEST) instruction is input to the device, the boundary scan register is set to allow simultaneous observation of signals input to the test cells and control of signals output from the test cells. Simultaneous control and observation is achieved by the design of the SCOPE cells. Each SCOPE cell contains two memories (flip-flops), one to observe input data and the other to control output data. During EXTEST or INTEST the TAP receives external input causing the boundary scan register to capture data on the CK, OC, IN, and REG output signals. After the data are captured, the TAP receives input to shift the captured data out via the TDO output. While captured data are shifted out, the next test control pattern to be applied from the boundary scan register is shifted in via the TDI input. The outputs from the boundary scan register are not allowed to change until the shift operation is complete and the TAP receives input to apply a new test control pattern from the boundary scan register outputs. This procedure of capturing input data, shifting the boundary scan path to extract captured data and load new test control data, followed by applying the new test control data from the boundary scan register outputs, is repeated a required number of times to perform an EXTEST or INTEST operation.

When a tristate and bypass (TRIBYP) test instruction is input to the device, the outputs are placed in a high-impedance state. This instruction is designed primarily to facilitate a blend of in-circuit and boundary scan testing. By disabling the outputs of the device, an in-circuit tester can drive the inputs of another device coupled to the outputs of the octals without damaging the octal's output buffers. While this instruction is in effect, the bypass register is selected to provide a minimum data register scan length through the device.

When a set and bypass (SETBYP) test instruction is input to the device, the boundary outputs are set to a prescanned combination of logic 1's and 0's. This instruction allows the boundary test cells to output a prescanned control pattern to the REG inputs, tristate buffers, and device outputs (OUT). The SETBYP instruction allows placing the octal device in a preferred static input and output state while testing of neighboring components is being performed. While this instruction is in effect, the bypass register is selected to provide a minimum data register scan length through the device.

To support a boundary BIST approach, a run boundary test in test mode (RBTTM) instruction was developed for SCOPE devices. The RBTTM instruction executes the boundary BIST operation setup by control-bit settings in the BCR of Figure 4. The control bits in the BCR must be set before inputting the RBTTM instruction. The four types of RBTTM instructions included in the SCOPE octal devices are 16-bit Parallel Signature Analysis (PSA) of the IN inputs, 16-bit Pseudo-Random Pattern Generation (PRPG) from the OUT outputs, simultaneous PSA of IN inputs and PRPG of OUT outputs, and simultaneous sample of IN inputs and toggle of OUT outputs.

During the 16-bit PSA RBTTM instruction, the 8-bit TCR1 and TCR2 boundary sections are linked to form a single 16-bit Linear Feedback Shift Register (LFSR). The parallel inputs to TCR1 are enabled to accept data from the IN bus, and the parallel inputs to TCR2 are disabled. In this configuration, TCR2 acts as an 8-bit LFSR extension to TCR1. During test, the parallel inputs from the IN bus are compressed into the 16-bit LFSR on the rising edge of TCK. Linking TCR1 to TCR2 allows the SCOPE octal to receive an extended sequence of 8-bit patterns from the IN bus. At the end of the PSA operation, the 16-bit signature can be shifted out of TCR1 and TCR2 for inspection. While TCR1 and TCR2 are collecting the signature, the outputs of TC1 and TC2 remain in their present state. TC2 can be set to enable or disable the OUT buffers during the test.

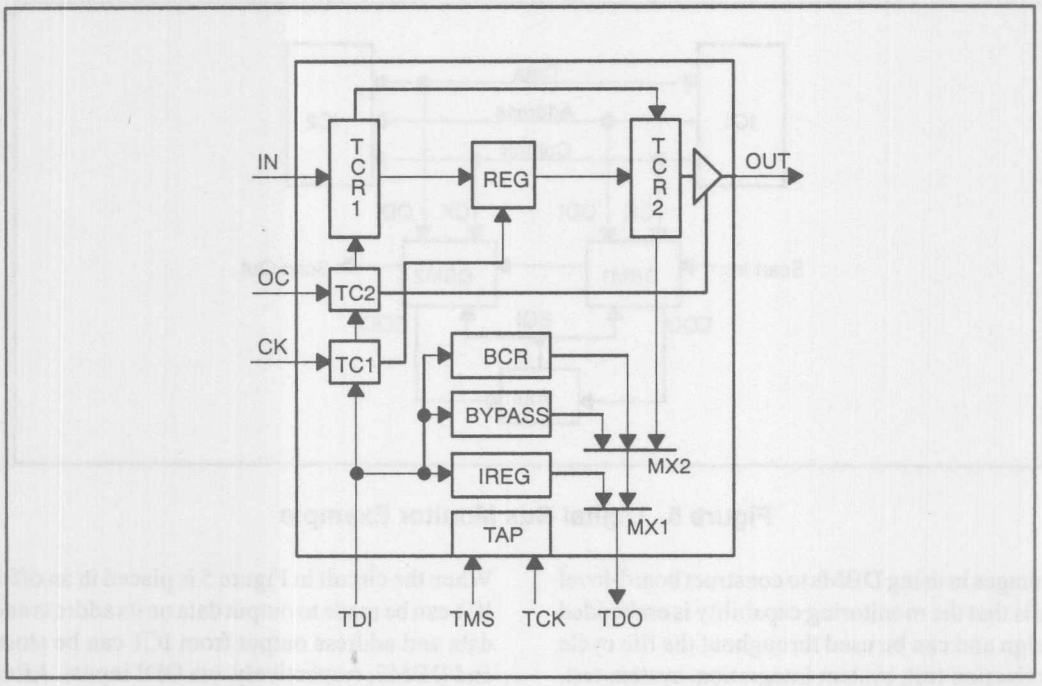


Figure 4. SCOPE Octal Architecture

During the 16-bit PRPG RBTTM instruction, the 8-bit TCR1 and TCR2 boundary sections are linked to form a 16-bit LFSR, as described in the 16-bit PSA test. During the 16-bit PRPG test operation, both parallel inputs to TCR1 and TCR2 are disabled so that both act only as LFSRs. During test, the parallel output from TCR2 drives pseudorandom patterns to the OUT bus on each falling edge of TCK. By linking TCR1 and TCR2, an extended set of pseudorandom pattern sequences is produced. Since the width of the OUT bus is 8 bits, individual patterns will be repeated during every 256-pattern output sequence. However, the test circuit will produce 256 sets of unique 256-pattern output sequences. During this test, TC2 must be set to enable the OUT buffers.

During the simultaneous PSA and PRPG RBTTM instruction, TCR1 and TCR2 operate as two separate 8-bit LFSRs. The parallel inputs to TCR1 are enabled to accept data from the IN bus, and the parallel inputs to TCR2 are disabled. During test, TCR2 outputs pseudorandom patterns to the OUT bus on the falling edge of TCK and TCR1 compresses input data from the IN bus on the rising edge of TCK. Combinational logic residing in the external path between the OUT and IN buses can be tested quickly using this instruction. During this test, TC2 must be set to enable the OUT buffers.

During the simultaneous sample and toggle RBTTM instruction, TCR2 outputs alternating data patterns to the OUT bus on the falling edge of TCK, and TCR1 accepts data input from the IN bus on the rising edge of TCK. By adjusting the frequency of the TCK, this test can be used to measure propagation delays through external logic residing between the OUT and IN buses. During this test, TC2 must be set to enable the OUT buffers.

SCOPE octal ICs will be available in 24-pin DIP and SOIC (Small Outline Integrated Circuit) or 28-pin LCC packages. The parts are designed in bipolar complementary metal-oxide semiconductor (BiCMOS) technology and provide 48/64 mA source/sink output buffers. The functional performance of the SCOPE octals approach 74F speeds, and the maximum scan and BIST clock rate is 20 MHz.

SCOPE Digital Bus Monitor (DBM) IC

To extend board-level testing, a Digital Bus Monitor (DBM) IC is being designed. The DBM can be included in board designs to provide a method of monitoring embedded digital signal paths between ICs. The DBM is capable of monitoring digital signal paths while the board circuitry is either on-line and operating normally or is in an off-line test mode. The benefit of on-line monitoring is that it can be used to reveal timing-sensitive and/or intermittent failures that are otherwise undetectable without the use of external test equipment and mechanical probing fixtures.

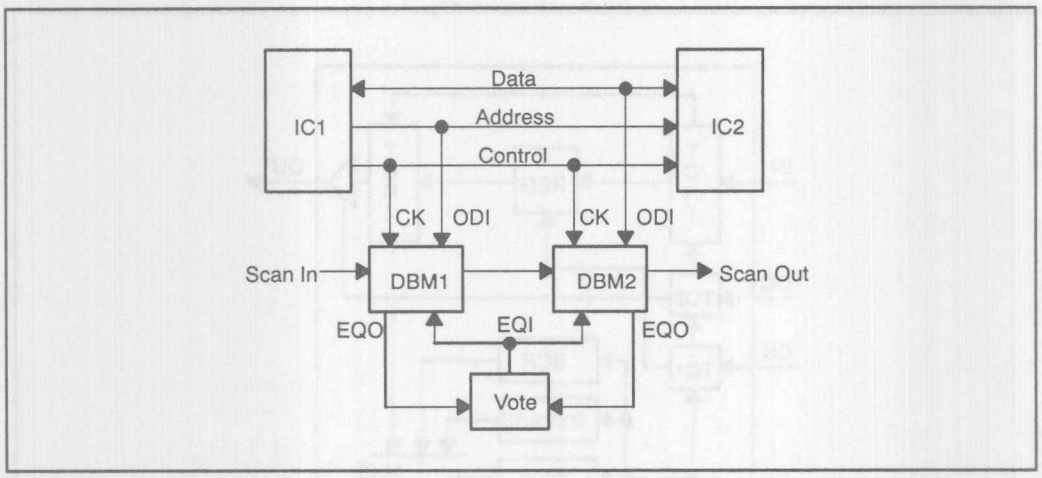


Figure 5. Digital Bus Monitor Example

One of the advantages in using DBMs to construct board-level BIST structures is that the monitoring capability is embedded in the board design and can be used throughout the life cycle of the board production test, system integration, system test, real-time diagnostics, and field support and maintenance. Another advantage in using DBMs is that it does not significantly impact the performance of the board circuitry. Since the signals to be monitored do not pass through the DBM but are only input to the DBM, no significant performance penalty is paid when using these devices.

In the example DBM application shown in Figure 5, two ICs operate together via address, data, and control interface paths to perform a desired function. In normal operation, IC1outputs address and control to IC2 to pass data between the two ICs. DBMs 1 and 2 are included in the circuit for address and data bus monitoring. The DBMs are connected via the 1149.1 four-wire scan bus and the SCOPE two-wire event qualification bus. The address and data bus signals to be monitored are input to the DBMs via observability data inputs (ODIs). The control outputs from IC1 are input to the DBMs via clock inputs (CK) to allow the DBM to operate synchronous with the circuit during on-line monitoring.

The test circuitry residing behind the ODI input pins consists of a Random-Access Memory (RAM) buffer, and a test cell register. The memory buffer provides storage for multiple ODI input patterns. The test cell register operates as either an 1149.1 boundary scan register or PSA register. The memory buffer and test-cell register can be operated together or separately, as required for a particular test operation. The ODI inputs to the test-cell register can be masked individually to allow diagnosing which input or groups of inputs caused a multiple-input PSA operation to fail.

When the circuit in Figure 5 is placed in an off-line test mode, IC1 can be made to output data on its address and data bus. The data and address output from IC1 can be stored into DBM1 and DBM2, respectively, via ODI inputs. After the data have been stored, they can be shifted out for inspection via the 1149.1 scan path. Similarly, IC2 can be made to output data on its data bus, to be stored and shifted out for inspection by DBM2. In the off-line test mode, control to store data and operate the scan path is input via the 1149.1 test bus.

When the circuit shown in Figure 5 is on-line and functioning normally, the DBMs can continue to monitor the data and address bus paths using an internal Event Qualification Module (EQM) resident in each DBM IC. During on-line monitoring, the EQM outputs control to store the data appearing on the DBM's ODI inputs. The EQM operates synchronous to the control signals input to the DBM's CK inputs. To determine when to store data, the EQM includes comparator logic that can match the data appearing on the ODI inputs against a predetermined expected data pattern or set of expected data patterns. The compare operation performed on each ODI input can be masked individually to eliminate input signals not required for event qualification. The EQM has protocols allowing it to perform different types of event-qualified monitoring operations. The type of monitor operation to be performed (RAM storage, PSA, sample) determines the protocol type used.

To expand the event-qualification capability, multiple DBMs or other SCOPE products incorporating EQMs can be connected via the two-wire event-qualification bus to allow qualification of a test operation to be distributed over a range of devices. During expanded event qualification, each DBM operates to output a match condition on its Event-Qualification

Output (EQO) pin. The match signals from multiple DBMs are combined via a voting circuit to produce a global match signal. The global match signal is input to each DBM via an Event-Qualification Input (EQI) signal. When a global match signal is received by the DBMs, the internal EQM initiates a test monitor operation. In some instances, it may be required to qualify a monitor operation further using external signals. In this case, the external signals are input to the voting circuit to allow finer resolution as to when a monitor operation is performed.

The SCOPE DBM ICs will be available in 28-pin PLCC or LCC packages. The devices will have a 16-bit ODI input bus. The internal RAM will have storage capacity for at least 1K patterns. The PSA capabilities of the DBMs will be cascadable to allow constructing PSA registers in multiples of 16 bits (i.e., 16, 32, 48 . . .).

Conclusion

The 1149.1 standard provides the framework for a structured test approach to eliminate the ad hoc techniques used in the past. The products previewed in this paper provide a starting point from which the realization of 1149.1 may begin. TI will continue to support and develop products for IEEE standards such as 1149.1 to provide the industry with improvements over existing techniques.

References

[1] *JTAG/IEEE 1149.1 Standard.*

[2] L. Whetsel. *A Proposed Standard Test Bus and Boundary Scan Architecture*, IEEE International Conference on Computer Design, Rye Brook, NY, October 3–5, 1988, pp. 330–333.

[3] F. Beenker. *Systematic and Structured Methods for Digital Board Testing*, Proceedings IEEE International Test Conference, 1985, pp. 380–385.

[4] C. Maunder and F. Beenker. *Boundary Scan: A Framework for Structured Design for Test*, Proceedings IEEE International Test Conference, 1987, pp. 714–723.

[5] M.M. Pradhan, R.E. Tulloss, F.P.M. Beenker, and H. Bleeker. *Developing a Standard for Boundary Scan Implementation*, Proceedings International Conference on Computer Design, Rye, New York, 1987, pp. 462–466.

[6] L. Whetsel. *Modular ASIC Test Cells for Boundary Scan Testing Applications*, Proceedings Government Microcircuit Applications Conference, 1989.

[7] *SCOPE Cell Design Manual*, Local Texas Instruments Sales Office.

[8] S. Vining. *Tradeoff Decisions Made for an 1149.1 Controller Design*, Proceedings IEEE International Test Conference, 1989.

[9] *SCOPE Octal Data Sheets*, Local Texas Instruments Sales Office.

[10] A. Halliday, G. Young, and A. Crouch. *Prototype Testing Simplified by Scannable Buffers and Latches*, Proceedings IEEE International Test Conference, 1989.

Modular Application-Specific Integrated Circuit (ASIC) Test Cells for Boundary Testing Applications

by Lee Whetsel

This paper was presented at the Government Microcircuit Applications Conference, 1989.

Abstract

Test techniques such as input/output (I/O) boundary scan, internal scan, and Built-In Self-Test (BIST) can improve an integrated circuit's ability to be tested, as well as the testing of the board it is mounted on. In many cases, the benefits of including test features in integrated circuits used in military and high-end commercial products extend beyond testability and into other key areas such as system integration, maintainability, and field support.

Introduction

To support a structured ASIC design-for-testability approach, a library of modular test cells has been developed. The library includes test cells required to implement the boundary-scan architecture and four-wire test bus described in the Joint Test Action Group (JTAG)/1149.1 IEEE standard. The library also contains test cells with added features to support an extended 1149.1 test architecture referred to as System Controllability, Observability, and Partitioning Environment (SCOPE). This paper describes the attributes of each test cell type and gives insight into how their test functions can be used at the boundary of ICs to facilitate chip- and board-level testing.

IEEE 1149.1 Overview

The 1149.1 IEEE standard provides the industry with an IC-level test framework consisting of a four-wire Test Access Port (TAP) controller and related scan-path architecture, as shown in Figure 1. The TAP controller receives external control input via Test Clock (TCK) and Test Mode Select (TMS) signals and outputs internal control signals to the internal scan paths.

The scan-path architecture consists of a single serial instruction register and two or more serial data registers. The two required data registers are a boundary-scan register and a 1-bit scan bypass register. The instruction and data registers are connected in parallel between a serial Test Data Input (TDI) signal and serial Test Data Output (TDO) signal. The TDI input is connected directly to the serial inputs of the instruction and data registers. The TDO output is connected via multiplexer 1 to either the serial output of the instruction or data registers. The selection control for multiplexer 1 comes from the TAP controller. Since multiple data registers can be used, multiplexer 2 is required to route a selected data register's serial output into multiplexer 1 to drive the TDO output. The selection control for multiplexer 2 comes from the instruction register.

When boundary testing is not being performed, the boundary-scan register is transparent, allowing input and output signals to pass freely to and from the IC logic. However, during boundary testing, the boundary-scan register disables the normal flow of input and output signals to allow the boundary signals of the IC to be controlled and observed via scan operations. The 1149.1 standard details how multiple ICs incorporating this scan architecture can operate together to perform wiring interconnect tests, as well as scan-operated tests of combinational logic residing between the boundaries of ICs in a circuit.[1,2,3,4]

SCOPE Cell Library

The test cell library contains predesigned implementations of the cells required to implement the 1149.1 standard. Test cells also are provided to implement the extended test capabilities defined in the SCOPE architecture. The library includes a TAP controller, instruction register cells, bypass register cells, and a family of modular boundary test cells for both unidirectional- and bidirectional-signal applications.

Table 1. SCOPE Boundary Cell Library

Cell Name	Cell Description
TSG00	Basic Unidirectional SCOPE Cell
TSG01	TSG00 with Signature Analysis Logic
TSG02	TSG01 with Programmable Polynomial Taps
TSG03	TSG00 with Maskable Compare Logic
TSG04	TSG01 with Maskable Compare Logic
TSG05	TSG02 with Maskable Compare Logic
TSB00	Basic Bidirectional SCOPE Logic
TSB01	TSB00 with Signature Analysis Logic
TSB02	TSB01 with Programmable Polynomial Taps
TSB03	TSB00 with Maskable Compare Logic
TSB04	TSB01 with Maskable Compare Logic
TSB05	TSB02 with Maskable Compare Logic

SCOPE boundary test cells provide a range of test capabilities to allow the ASIC designer to implement the level of test required to meet IC- and system-test needs[5]. Although each SCOPE cell offers a different blend of test capability, all are designed around a common hard-macro base cell to provide a consistent, fundamental mode of operation. Table 1 lists the current types of SCOPE boundary cells that can be used to construct the boundary-scan register of Figure 1. The remainder of this paper describes the test capabilities offered by each SCOPE cell type.

TSG00 and TSB00 SCOPE Cell Description

The TSG00 and TSB00 cells provide the basic test capabilities required to implement an 1149.1 boundary-scan register. These cells can be distributed around the periphery of IC designs and linked together serially to allow individual control and observation of input and output boundary signals via the four-wire test bus in Figure 1. The control for operating these cells in a boundary-scan test mode is issued from the TAP controller, while the control for placing the cells in a test or normal mode comes from the instruction register of Figure 1.

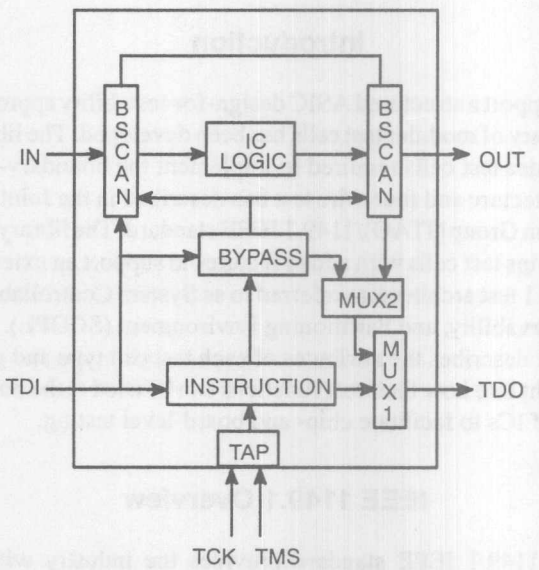


Figure 1. 1149.1 Architecture

When in normal mode, the cells are transparent and allow boundary signal data to enter and exit the cells unobstructed. When in test mode, the cells form a partition at the IC boundary, allowing data entering the cell to be observed while con-

trolling data output from the cell. To achieve simultaneous observation of input data and control of output data, the cells incorporate two memory elements (flip-flops). The first memory operates to load and shift out the data signal input to the cell, while the second memory is used to maintain a desired test control output from the cell.

In addition to their 1149.1 test capabilities, these cells offer a toggle mode of operation. The toggle mode allows the cells output to alternate between opposite logic levels at the rate of the TCK input. During IC-level testing, the toggle mode simplifies the testing and measurement of output buffer low-to-high and high-to-low switching times. At the board level, the toggle mode is useful in testing for gross timing delays between a driving and receiving IC. If desired, the toggle mode of the TSG00 and TSB00 cells can be further used to construct sections of a boundary-scan register capable of generating binary count-up-or-down patterns at the rate of the TCK input. For instance, the inclusion of binary-counting capabilities in the address-boundary section of a processor IC simplifies memory access during both control store upload and testing operations.

To provide implementation flexibility, the cells are designed to allow either synchronous or asynchronous modes of operation. In the synchronous mode, the cells operate from a free-running clock and control inputs to cause the cell to transition between its four operational states (hold, load, shift, and toggle) to perform a test operation. In the synchronous mode, the hold state is used when no test action is required. In the asynchronous mode, the cells operate from a gated clock and control inputs to make the operational-state transitions required to perform a test operation. In the asynchronous mode, the clock is gated off when no test action is required.

The TSB00 differs from the TSG00 only in that the TSB00 contains additional multiplexers and control signals to accommodate bidirectional boundary-signal types. The advantages of using one TSB00 cell on a bidirectional signal versus using two TSG00s are a reduced gate count and boundary-scan register bit length. The reduced gate count is achieved by simply adding two multiplexers for input and output signal routing to a TSG00 cell, enabling the TSG00 to function as both the input and output cell on a bidirectional signal. Since the TSB00 only requires a single flip-flop memory to test both the input and output paths of a bidirectional signal, the scan-path bit length for bidirectional signals in the boundary-scan register is reduced by one-half.

TSG01 and TSB01 SCOPE Cell Description

The TSG01 and TSB01 cells provide all the test capabilities offered in the TSG00 and TSB00 cells. In addition, the TSG01 and TSB01 cells include additional test circuitry to enable the cells to act as bit-slice signature analysis elements. By substituting the TSG01 and TSB01 cells for the TSG00 and TSB00 cells, the designer easily can incorporate Parallel Signature Analysis (PSA) capabilities into particular sections of the boundary-scan register that are receptive to data-compression techniques. PSA testing offers two improvements over 1149.1 boundary-scan testing: a reduction in the time to test and the capability to test at speed.

During PSA testing, groups of TSG01 or TSB01 cells operate together as Linear Feedback Shift Registers (LFSRs) to compress multiple data inputs into a signature, as shown in Figure 2. After the PSA test is complete, the signature can be shifted out of the LFSR and compared to a known good signature. Because the data inputs can be compressed at speed, timing-sensitive failures that may be undetectable, using boundary scan alone, are detected using this approach.

To provide a method of diagnosing the reason a multiple-input PSA operation failed, the TSG01 and TSB01 cells incorporate a data input masking capability as shown in Figure 2. The masking feature allows the cell's data input to be disabled so that it has no impact on the signature being taken. While masked, a cell forces its data input to a logic zero during the PSA operation, regardless of the data actually being input to the cell. If a multiple-input signature fails, the test can be repeated with selected cell inputs masked off to produce a different signature that reflects the compression of only the non-masked cell inputs. Using this approach, it is possible to isolate one or more single-input signature failures that caused the failure in the multiple-input signature.

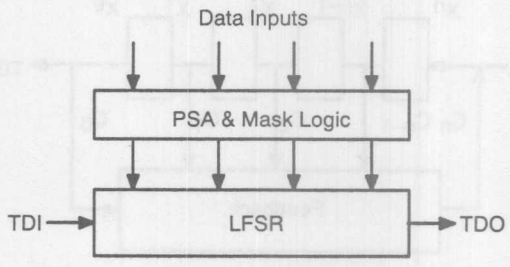


Figure 2. Maskable PSA Example

TSG02 and TSB02 SCOPE Cell Description

The TSG02 and TSB02 cells combine the test capabilities offered in the TSG01 and TSB01 cells with a programmable polynomial tap circuit consisting of an exclusive-OR gate and switching logic. In LFSR applications, such as PSA and Pseudo-Random Pattern Generation (PRPG), it may be desirable to have programmable control of the feedback connections between the register cells and an exclusive-OR network.

By providing a means to program these feedback connections, it is possible to modify the behavior of LFSR circuits, allowing them to produce different output pattern lengths and sequences.

The programmable polynomial tap circuitry of the TSG02 and TSB02 cells allows the data stored in the cells to be included in or excluded from the feedback network as required to implement a feedback connection (or characteristic polynomial) suitable for a particular LFSR test function. There are two types of LFSR implementations: internal and external. The internal type of LFSR includes an exclusive-OR gate in the serial path between each cell in the register. The external type places the exclusive-OR feedback gating external to the serial path and provides input to the feedback logic from the serial output of each cell in the register. While the TSG02 and TSB02 cells can be used to implement either type of LFSR, this paper describes their use in external-type LFSR implementations.

In the external-type LFSR example shown in Figure 3, a series of these cells are linked together through the serial scan path of each cell and by the feedback tap connections made to each cell stage. The ordering of the polynomial feedback taps for an external LFSR start by assigning the output of the last cell in the series a value of X^0 and progressing up to a value of X^{n-1} for the output of the first cell in the series. The input to the first cell in the series is driven by the output of the feedback network and assigned the value X^n.

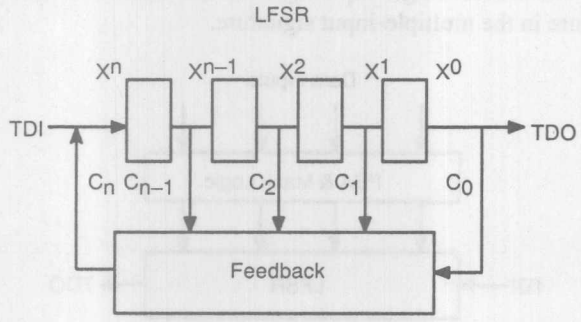


Figure 3. LFSR Example

Using this ordering convention, external-type LFSR feedback tap connections can be expressed in a characteristic polynomial equation shown in Equation (1). In the equation, n represents the number of cell elements in the LFSR, and the coefficients C^n, C^{n-1}, ... C_0 indicate whether a tap connection exists between the feedback network and a corresponding cell element n, n–1, ... 1. If a C term is set to a zero, no connection exists between the feedback network and related LFSR cell and the term drops out. However, if a C term is set to a one, the cell is coupled to the feedback network and the term remains in the following equation:

$$C_nX^n + C_{n-1}X^{n-1} \ldots + C_1X^1 + C_0X^0 \quad (1)$$

For example, a characteristic polynomial producing a maximum-length pattern sequence for an LFSR consisting of eight cells is shown in Equation (2). The coefficients for terms 8, 6, 5, 4, 0 are set to one, causing them to be present in Equation (2), while the coefficients for terms 7, 3, 2, 1 are set to zero causing them to be absent from the following equation:

$$X^8 + X^6 + X^5 + X^4 + 1 \quad (2)$$

An example implementation of an external-type LFSR implementing this characteristic polynomial is shown in Figure 4.

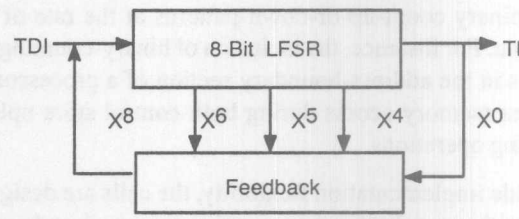


Figure 4. 8-Bit LFSR Example

Typically, LFSR circuits used in testing applications implement polynomials that produce a maximum-length pattern sequence of 2^{n-1}. Polynomials producing maximum length may require only a minimum number of connections between the feedback network and the cells in the LFSR, as illustrated in Equation (2) and Figure 4. Tables of predetermined maximum length polynomials may be referenced from existing sources[6]. For any given length LFSR, multiple polynomials may exist that will achieve a maximum-length operation. However, each unique polynomial generates a different pattern sequence en route to reaching the maximum length.

In Figure 3, the programmable polynomial tap circuitry in the TSG02 or TSB02 cells can be set to make or break the tap connections to the feedback network. This ability to program the tap connections allows the designer complete freedom to modify the behavior of boundary-resident LFSRs to adjust the circuit's operation to a particular boundary test need.

In PRPG applications, the ability to program the feedback tap connections allows modifying the pattern length and sequence produced. This ability to modify the pattern length and sequence is sometimes required to improve the fault detection of circuits that exhibit a sensitivity to a particular pattern stimulus sequence.

In PSA applications, the ability to program the feedback tap connections allows for concatenating a series of cells together to produce an optimized LFSR length suitable for a particular data-compression operation. As the cell length of an LFSR is modified, a new feedback tap polynomial is required to produce a maximum-length mode of operation. The ability to lengthen the number of cells in an LFSR increases the number of input patterns that can be received and compressed into a signature without a corresponding increase in the probability of signature aliasing.

TSG03,04,05 and TSB03,04,05 SCOPE Cell Description

The TSG03,04,05 and TSB03,04,05 cells combine the test features offered in the TSG00,01,02 and TSB00,01,02 cells with a maskable compare logic section. In addition to their boundary controllability and observability attributes, these SCOPE cells provide a method of comparing incoming and outgoing boundary signals against an expected pattern. When a match condition occurs between an expected data bit and a boundary input or output signal, the cells output a compare signal to indicate the match. By inputting the individual compare outputs from each SCOPE cell into a combining circuit, a global compare signal can be produced, indicating the occurrence of a matching condition on a range of input and output cells.

In addition, the compare logic includes a masking feature that can be used to force a true compare output from the cells, regardless of whether a match has occurred. This masking capability provides a method of assigning "Don't Care" condition attributes to selected boundary input and output signal pins when they are not involved in producing the global compare signal.

The ability to detect the occurrence of boundary conditions can be used in many different ways. One use of this boundary comparison technique is to output a match point indicator from an IC telling a test controller when a particular boundary condition occurs. Upon receiving the match point, the test controller can initiate a test operation on the IC issuing the match point. The SCOPE architecture uses this boundary comparison technique to initiate some of its additional test functions, such as on-line boundary sampling and signature analysis. Future SCOPE products will include these and other types of features to support board-level at-speed testing and hardware/software integration efforts.

Conclusion

This paper describes the types of test cells offered in the SCOPE cell library. The modularity of the cells allows for the implementation of a variety of boundary test and built-in test features, supporting a range of ASIC test applications. Future additions to the SCOPE cell library are planned to provide other building-block cells for simplification of ASIC design for testability.

References

[1] *IEEE P1149.1 Draft 3 Standard Proposal.*

[2] L. Whetsel. *A Proposed Standard Test Bus and Boundary Scan Architecture*, IEEE International Conference on Computer Design, Rye Brook, NY, October 3–5, 1988, pp. 330–333.

[3] C. Maunder and F. Beenker. *Boundary Scan: A Framework for Structured Design for Test*, Proceedings IEEE International Test Conference, 1987, pp. 714–723.

[4] M.M. Pradhan, R.E. Tulloss, F.P.M. Beenker, and H. Bleeker. *Developing a Standard for Boundary Scan Implementation*, Proceedings International Conference on Computer Design, Rye, New York, 1987, pp. 462–466.

[5] *SCOPE Cell Design Manual.* Local Texas Instruments Sales Office.

[6] W.W. Peterson and E.J. Weldon, Jr. *Error Correcting Codes*, MIT Press, Cambridge, MA, 1972.

[7] D.K. Bhavsar and R.W. Heckelman. *Self-Testing By Polynomial Division*, 1981, IEEE Test Conference, pp. 208–216.

Partitioning Designs with 1149.1 Scan Capabilities

by Steve Altaffer

Introduction

Typical 1149.1 Scan Architecture Descriptions

Star vs. Scan path system scan designs addressed in IEEE 1149.1 use either a single scan path, employing a common mode line and daisy-chained data (scan path), or separate mode lines to individually controlled independent rings and common TDI/TDO signals (star). Figure 1 depicts each configuration. Each can have advantages over the other depending on applications and design requirements.

The reduced path lengths of a star configuration simplify the scan controller design since each scan path is shorter than if the entire system were connected on a single scan path. Although the same amount of data may be necessary to control the entire system, the overhead of tracking unused elements in the scan path for a particular test is reduced. The star configuration is usually recommended for systems that require individualized control over independent functions or shorter scan-path lengths to simplify the scan controller design. The star configuration also has an inherent fault tolerance as an advantage. Because each of the rings in a star configuration is isolated from the system, a fault within the scan path itself does not corrupt the entire system scan path as would be the case in a scan-path configuration. A multi-board system employing a scan-path configuration has the ability to control all of the elements in a scan path regardless of the board on which they are located. A system that is backplane bus intensive or has functions spread across board boundaries would most likely employ a scan-path configuration to simplify board boundary and functional testing.

Scan-Path Linker (SPL) and Scan-Path Selector (SPS) Overview

Theory of Operation

The scan ring family of devices (SPS and SPL) provide the ability to create a hybrid star-scan-path architecture in a system while maintaining the advantages of both. The devices can be used to bypass an entire scan path on a board or select up to four independent rings (SPL) on the board. A single scan path can be selected and scanned or a combination of rings can be linked together in series, depending on the scan ring support device being used. Figure 2 depicts the SPS as it would be connected in a system.

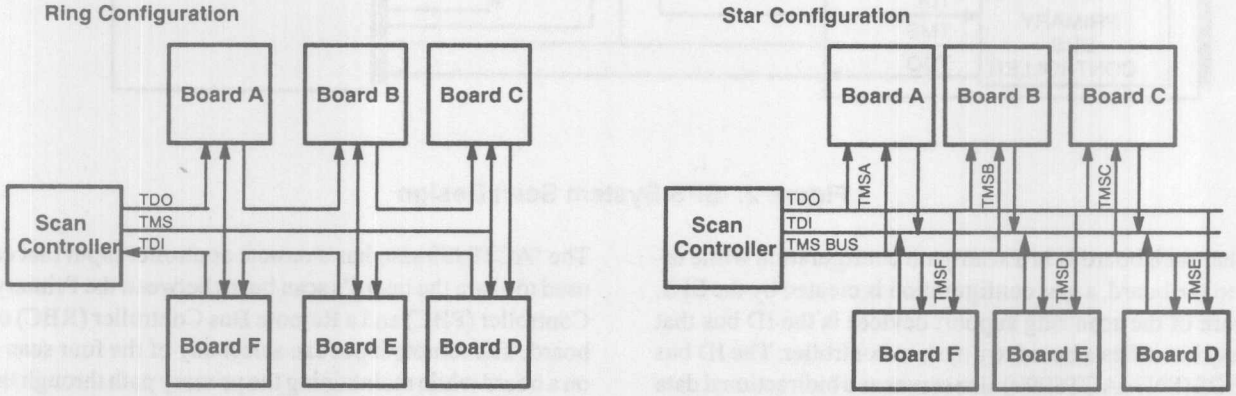


Figure 1. Ring and Star Configurations

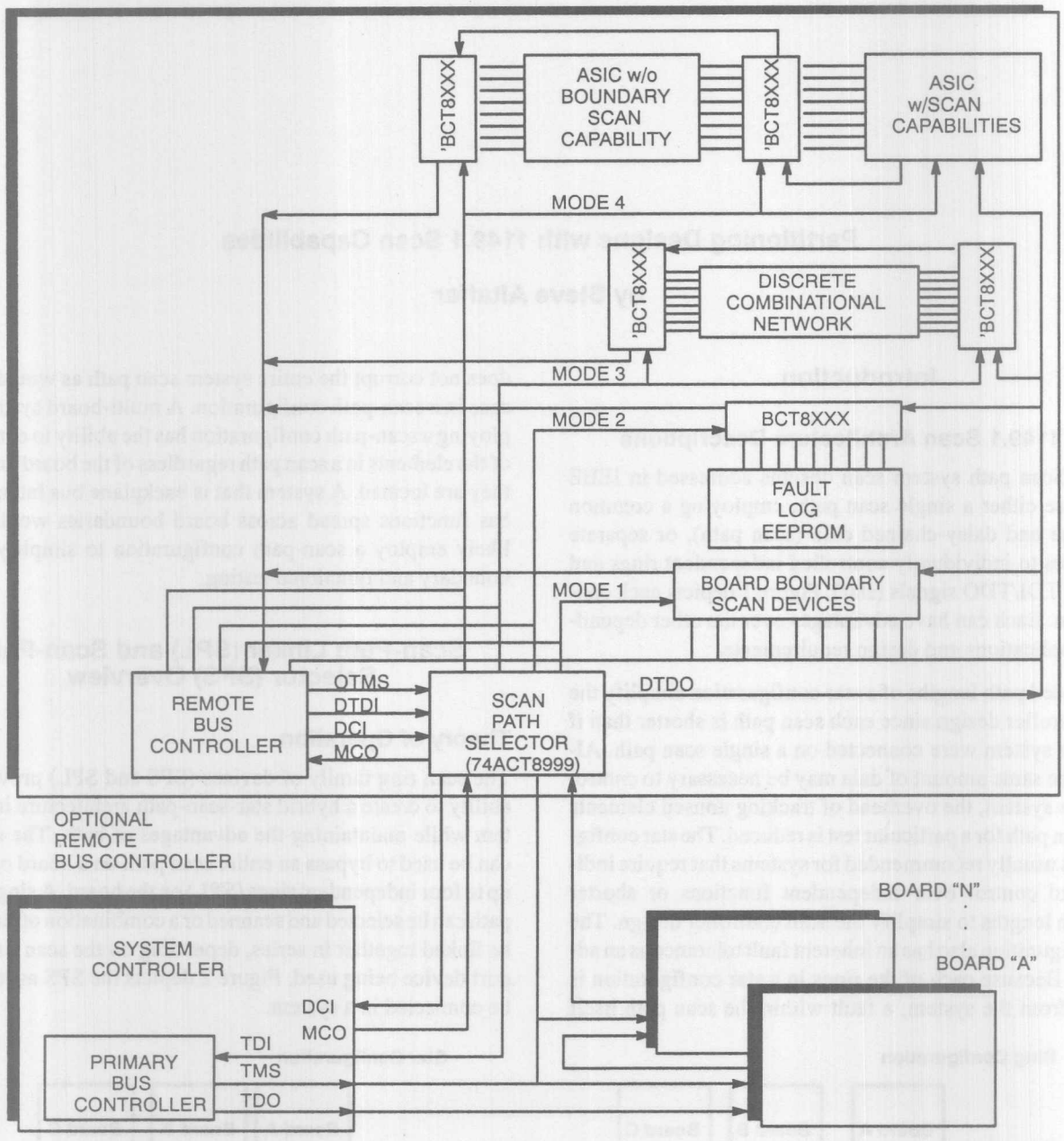


Figure 2. SPS System Scan Design

Note that each board is in a scan-path configuration while internal to the board, a star configuration is created by the SPS. A feature of the scan ring support devices is the ID bus that uniquely identifies a board to a system controller. The ID bus in the SPS (SN74ACT8999) also serves as a bidirectional data bus (BiD), capable of communicating with a remote bus controller or other on-board controller through the scan interface.

The 'ACT8999 also has a remote controller input that can be used to share the board's scan buses between the Primary Bus Controller (PBC) and a Remote Bus Controller (RBC) on the board. The remote input can select any of the four scan rings on a board while maintaining the primary path through the device to the rest of the system. In order to properly handshake with the board for remote-control handoff, ID bus operation,

or status/interrupt capabilities, a four-wire status bus is included in the devices. Those wires are labeled Master Condition Input (MCI), Master Condition Output (MCO), Device Condition Input (DCI) and Device Condition Output (DCO). The DCI pin also serves as the clock input to an 8-bit internal programmable counter. Properties such as the polarity of the clock, up/down counting, and latch-on-zero in the countdown mode are all programmable. The DCO pin can serve as the terminal count (MAX/MIN) output to allow cascading or interrupting the scan controller.

Select Register Operation

The SELECT data register is an 8-bit control register that determines the output of each of the DEVICE TEST MODE SELECT (DTMS) signals (2 bits per output). The select decoding for a single DTMS is shown in Table 1.

Table 1. Select Register Decoding

Select Bit MSB	LSB	DTMS Output
0	0	High (STRAP)
0	1	Low (IDLE)
1	0	OTMS *
1	1	TMS

* For devices without OTMS input, TMS is transferred to DTMS.

It is the same for each of the other DTMS outputs. For the case of a scan-path device not having an OTMS (remote) input, the device will route the primary TMS to the selected DTMS output. When an output or outputs are selected to convey TMS, the scan-path device waits until the primary TMS enters the IDLE state before multiplexing the TMS and TDI to the secondary scan path. This ensures proper state synchronization between the primary and board scan path(s) with respect to the TAP controllers on each. Any other means to "open" a secondary scan path would result in scan-path corruption and asynchroneity of data and control, and therefore is not allowed within the scan-path device hardware. This must be clearly understood by the software or firmware controlling the scan-path device so that the hardware is configured as the software believes it is. Referring to Figure 3, when a secondary scan path is opened, the TMS is driven directly to the output with internal delays only. Due to the routing of the data path directly from TDI to DTDO, it is necessary to sample the data on the rising edge of TCK and output to the secondary scan paths on the falling edge of TCK to comply with IEEE 1149.1 specifications. This creates an extra bit of data in the path whenever an external scan path is selected, but it is necessary to ensure data synchronization to the rings at high TCK frequencies. This sync bit will be present in all data and instruction scans and must be recognized by the scan bus controller whenever a secondary scan path is opened. The data from the scan path is then fed to the "normal" input of the scan-path device data or instruction registers and output to the primary scan path. For the SPL (SN74ACT8997), much the same data path is created.

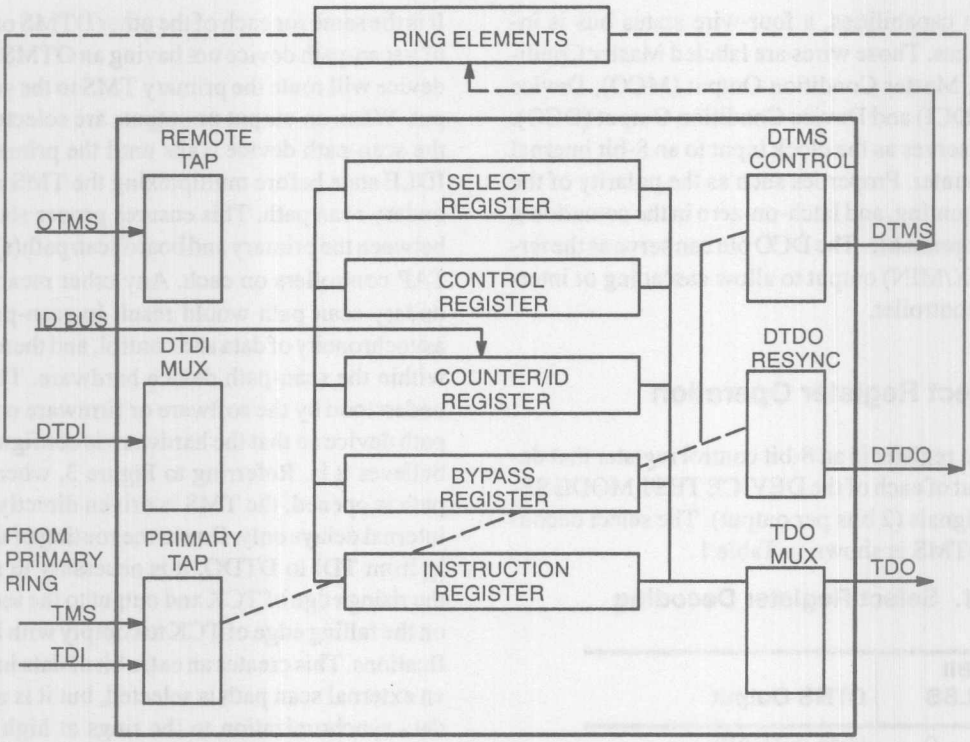


Figure 3. Scan-Path Selector (SPS) Scan-Path Ring with Ring Selected

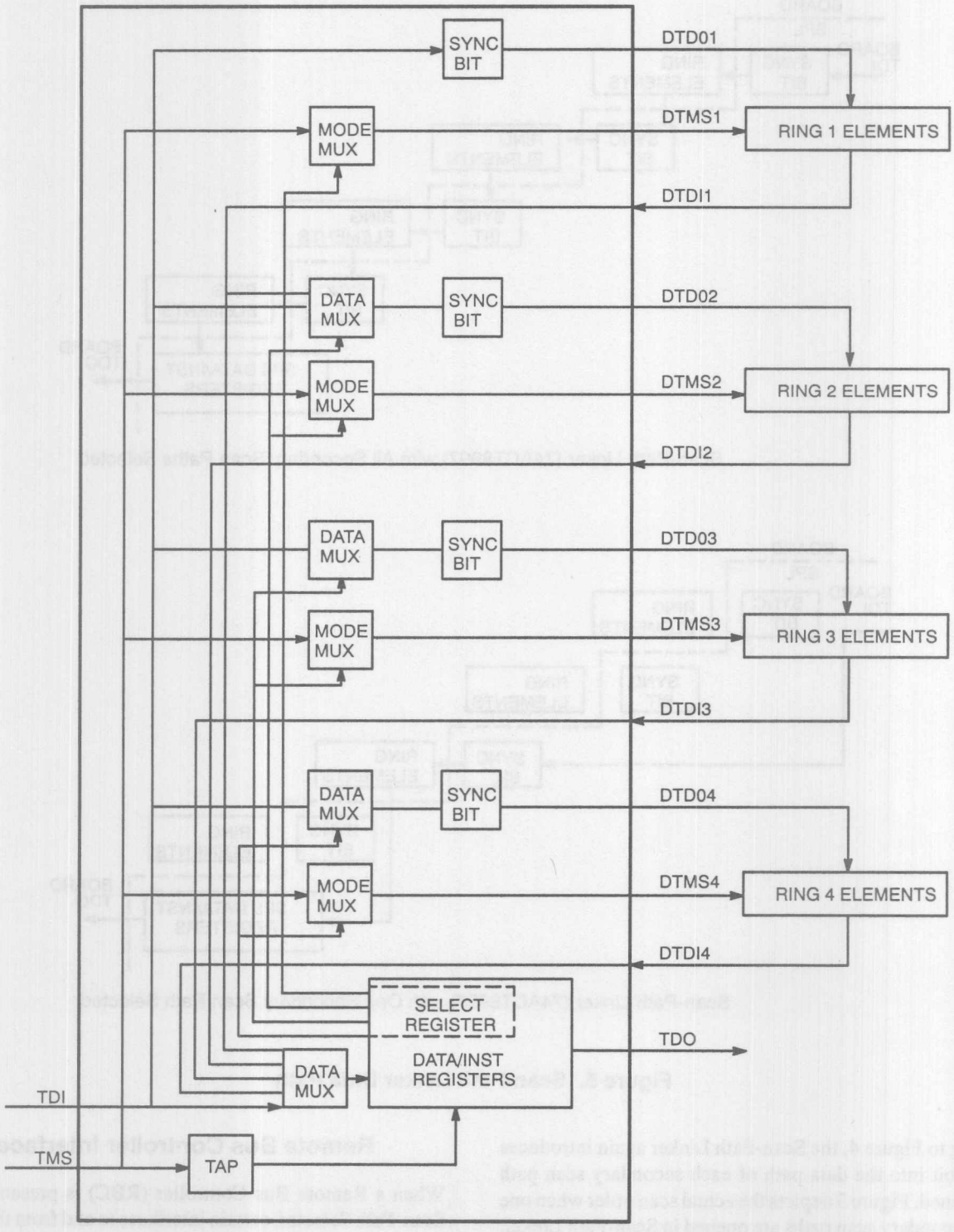
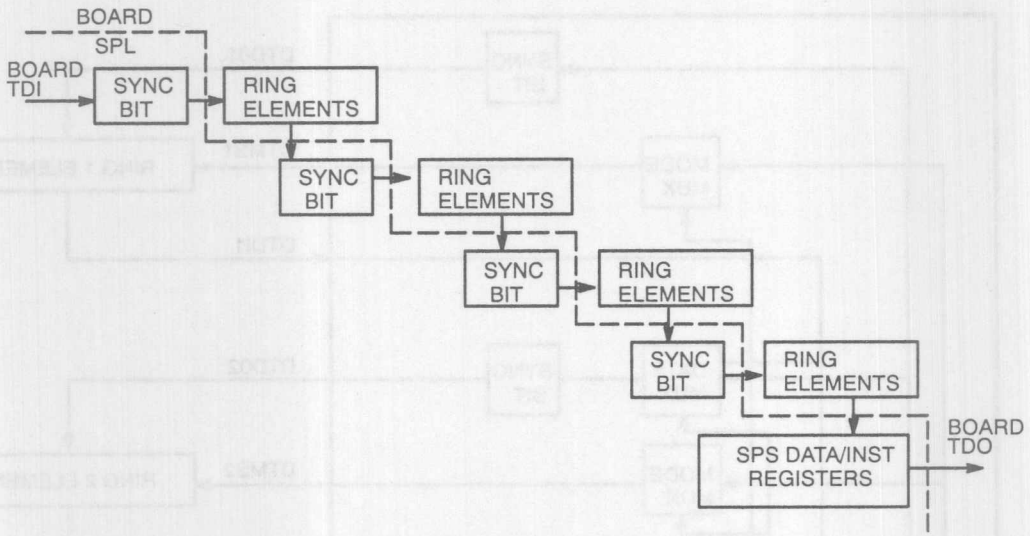
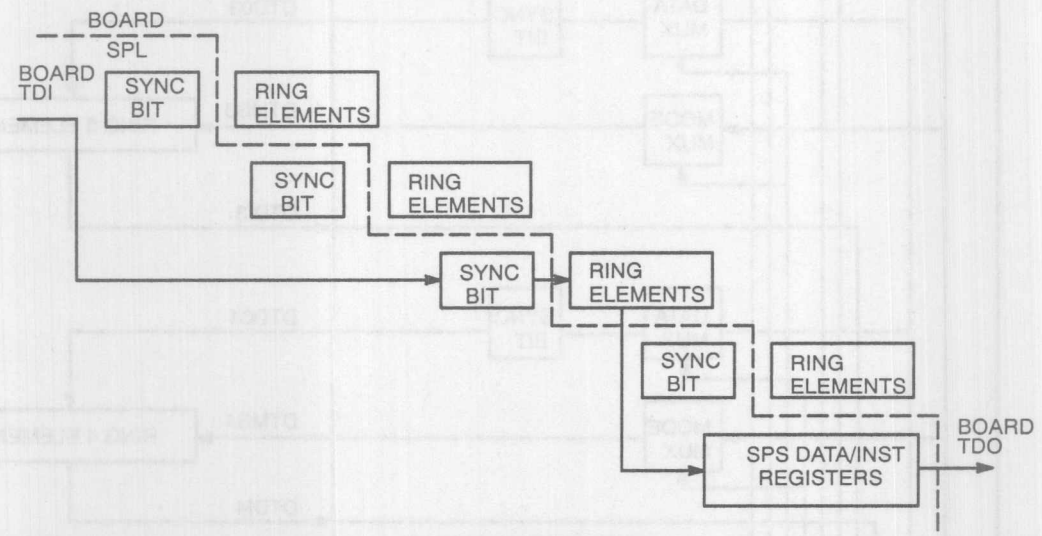


Figure 4. Scan-Path Linker (SN74ACT8997)



Scan-Path Linker (74ACT8997) with All Secondary Scan Paths Selected



Scan-Path Linker (74ACT8997) with One Secondary Scan Path Selected

Figure 5. Scan-Path Linker Data Path

Referring to Figure 4, the Scan-Path Linker again introduces an extra bit into the data path of each secondary scan path when opened. Figure 5 depicts the actual scan order when one or four secondary scan paths are opened in Scan-Path Linker.

Each of the sync bits are bypassed when a scan path is not selected and only the scan path or rings that are selected will introduce an extra bit into the path.

Remote Bus Controller Interfaces

When a Remote Bus Controller (RBC) is present with the Scan-Path Selector, certain interfaces to and from the SPS device need to be modified to ensure proper operation. The RBC inputs its TMS output to the OPTIONAL TMS (OTMS) pin of the SPS. When the SELECT register is loaded to output OTMS to the DTMS outputs, the CONTROL register must

also be loaded with the RBC Enable bit (RBCE) set TRUE to allow the RBC TMS direct control of the SELECT register. In this mode the Remote Test Access Port (RTAP) controls the shifting of data through the DTDI to the DTDO to the rings. When the RTAP is in control a dedicated instruction and bypass register become activated to emulate a true 1149.1 interface as seen by the RBC, as depicted in Figure 6.

The only valid instructions are 'scan the select register' or 'scan the bypass bit' during this mode of operation. The Primary Bus Controller (PBC) still has the ability to scan the other registers in the SPS that enables it to disable the RBC by setting the RBCE bit to FALSE should it become necessary. When connecting an RBC to the SPS the normal interfacing must be altered as shown in Figure 7.

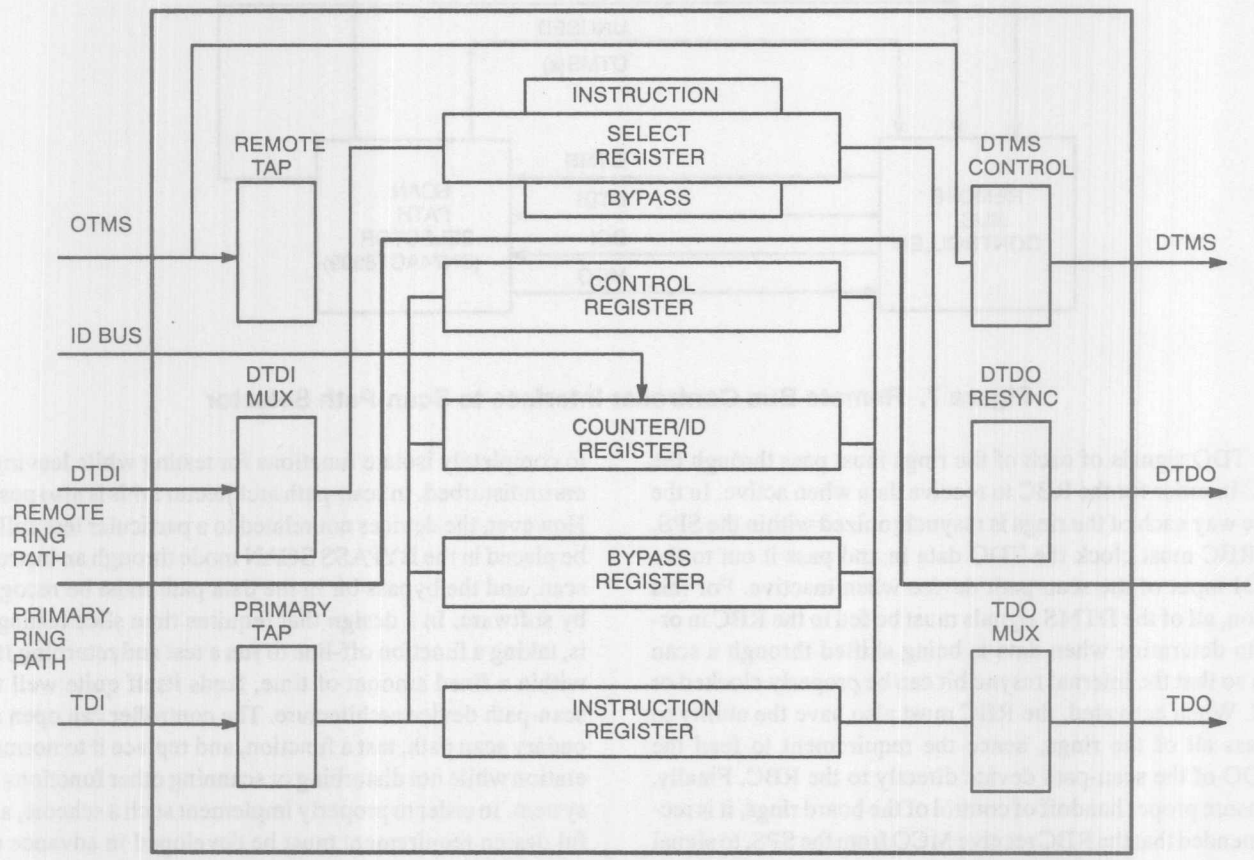


Figure 6. Scan-Path Selector (SN74ACT8999) with Optional (Remote) TMS Selected

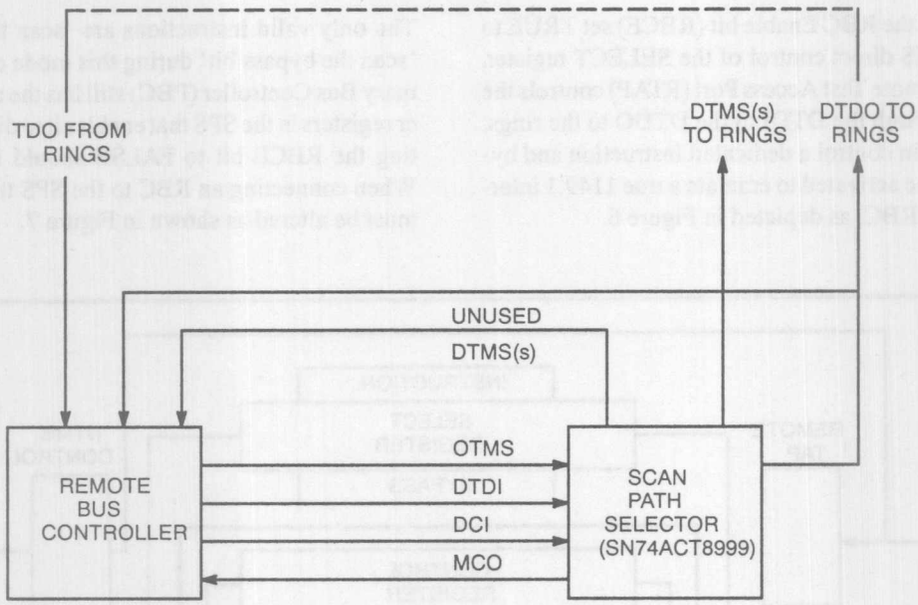


Figure 7. Remote Bus Controller Interface to Scan-Path Selector

The TDO signals of each of the rings must pass through the RBC in order for the RBC to receive data when active. In the same way each of the rings is resynchronized within the SPS, the RBC must clock the TDO data in and pass it out to the DTDI input of the scan-path device when inactive. For this reason, all of the DTMS signals must be fed to the RBC in order to determine when data is being shifted through a scan path so that the internal resync bit can be properly clocked or held. When activated, the RBC must also have the ability to bypass all of the rings, hence the requirement to feed the DTDO of the scan-path device directly to the RBC. Finally, to ensure proper handoff of control of the board rings, it is recommended that the RBC receive MCO from the SPS, to signal a grant from the PBC, and send DCI to the SPS, to indicate to the PBC that the RBC is currently using the board rings. This is also the handshake mechanism between the PBC and RBC for BiD bus transfers.

Partitioning 1149.1 Designs Using the Scan-Path Support Devices

System Level Partitioning and Considerations

Design Requirements

The ability to create a hybrid star/scan-path architecture using a scan-path device has many advantages over a single scan-path design. Scan designs can be partitioned in such a way as to completely isolate functions for testing while leaving others undisturbed. In scan-path architecture this is also possible. However, the devices not related to a particular test still must be placed in the BYPASS SCAN mode through an instruction scan, and the bypass bit in the data path must be recognized by software. In a design that requires time slice testing, that is, taking a function off-line to run a test and returning it back within a fixed amount of time, lends itself quite well to the scan-path device architecture. The controller can open a secondary scan path, test a function, and replace it to normal operation while not disturbing or scanning other functions in the system. In order to properly implement such a scheme, a careful design requirement must be developed in advance to account for any possible unexpected operations when exercising a function. For instance, an ASIC self test may inadvertently toggle its device outputs that in turn ripple to another function not under test. The inputs of the ASIC under test must be isolated so that the circuits feeding it do not effect the self-test. Avoiding inadvertent operation is probably the most common consideration but there are many others when implementing a time slice test scheme. Only through proper system definition and specification can all of the pitfalls be avoided. Another advantage of using a scan-path device is the ability to have a remote bus controller resident on the board to relieve the test burden of the primary bus controller. This creates a distributed test structure in which the PBC can command tests to be run autonomously on multiple boards and report status back. The relief realized at the system level is, of course, not

without cost. The development of board controller software will still be necessary as well as the handshaking and reporting structure between master and remote. This structure has a payoff throughout a product life cycle by exploiting tests during board and system integration, board production and depot testing as well as common module insertion. It can also reduce system test times by having all tests within a system running tests concurrently and reporting status instead of a single primary system executing tests in series.

Partitioning of Scan Paths

The proper partitioning of rings is the single most important factor in successfully reaping the benefits of scan-path device insertion into a scan design. Normally a board is functionally partitioned within a system by design. Within a board, functions also need to be partitioned for scan testing by placing test-related scan elements adjacent in the scan path. When using a scan-path device this is a requirement to enable efficient use of the device. If a test requires scanning of devices on separate rings of a scan-path device on a board, the controller must scan the devices to stimulate a test, scan the scan-path device SELECT register to place the stimulus secondary scan path in IDLE in order to hold the data, and open the response scan path, and then scan the results from the response scan path. This procedure becomes even more complex should the stimulus and response elements be mixed on separate rings. Should the test involve a dynamic stimulus and response, that is not static patterns, repeatability may not be possible. This fact arises due to the need to leave either the stimulus or response elements in IDLE (which implies running) while the other is being initialized. The time between scanning the first scan-path elements and the second could be variable or cause unknown data to be generated or sampled. Careful timing analyses can be performed to eliminate the problem but such an architecture is not recommended. Instead, the linkable scan-path device, SPL, should be used if scan elements for tests MUST reside on different rings for multiple uses, i.e., an element in RING1 is used for stimulus in a test with RING2, and response with elements on RING3. In most cases this is avoidable or optimization is possible to minimize the number of "inter-scan path" tests through proper partitioning.

How Much Scan

Another consideration for system-level designs is how much partitioning should be performed. Each scan-path device has four rings that can be used in a design, but not all of them have to be used. If the design does not lend itself to creating four separate rings, two or three can be used. This can create an open scan-path condition when an unused scan path is inadvertently selected. Great care must be taken during software development and design to avoid this possibility from occurring through constraint checking. If SPL is used, the DTDO and DTDI of unused rings can and should be tied together to at least create a path for data. Software must still detect the lack of any device(s) on the scan path that was opened, based upon instruction scan error checking or through examination of the scan-path device's SELECT register. This problem compounds itself when multiple scan-path devices exist within a system that have unused DTMS signals. Under such conditions the scan controller software will have to find the SELECT register contents of each scan-path device in the scan path to detect the error, determine the true state of the scan path, and correct the problem. The controller software becomes an issue in determining whether to use the scan-path device in a design. The obvious solution is to use all of the scan rings available on the scan-path device. Switching between multiple secondary scan paths for a test again becomes a problem and the SPL is recommended. If a remote controller is being used, all of the DTMS signals and the DTDO must be fed to it and can be used to detect the error condition, maintain the data-path integrity, and signal the PBC through the DCI-> DCO status signals of the condition.

Board Boundary Testing

When implementing scan-path devices into designs that contain board boundary scan elements to test backplanes or motherboards, two options exist for placing the boundary elements as depicted in Figure 8.

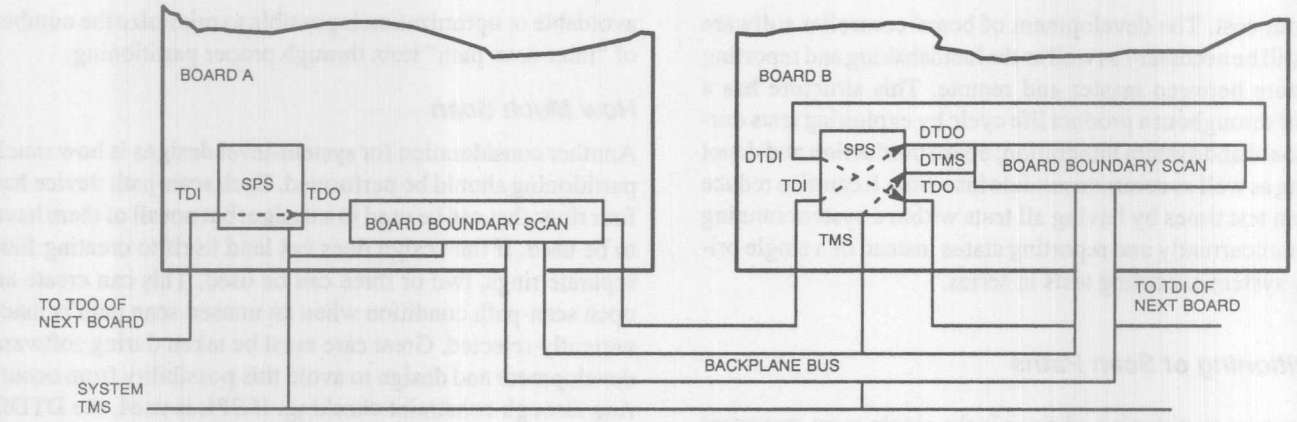


Figure 8. Board Boundary Element Scan-Path Options

Board A has the boundary elements placed on the primary scan path of the system. This would allow for two scenarios. First, if an RBC was also present on the board, the PBC could perform board boundary tests while the RBC performed board tests. Secondly, this implementation also allows selecting a scan path on the board and scanning the boundary elements to perform multiple tests concurrently. The disadvantage of this scheme is that the board is no longer isolated from the system by a single interface and some of the fault tolerance of the scan-path device architecture is lost. Board B places the boundary elements on a scan path of the scan-path device. This option retains all of the advantages of the scan-path device fault tolerance with the trade-off being the inability to run RBC concurrent tests if one is present in the design. In both cases board-to-board testing is still possible using the system scan-path architecture and the ability to open a scan path on each board.

Partitioning a Board Design – An Example

Memory Board Description

The example given is a simple memory board. It consists of a bus interface controller that could be an ASIC or discrete circuitry. The bus controller decodes address and control activity on the backplane and creates the necessary address and control signals required locally for memory and I/O cycles. The board also includes further decoding to create specific device selects for the memory and I/O arrays and any further control signal generation not performed within the bus controller. Finally, the board contains a memory array and I/O register array. The memory array could contain both static- and PROM-type devices.

Partitioning for Test

The buffers for the board are chosen from the 'BCT8XXX family of 1149.1-compatible octal devices ('BCT8244,8245,8373,8374). Referring to Figure 9, the placement of these devices partition the board into its three functions; bus controller, decode circuitry, and the memory and I/O space.

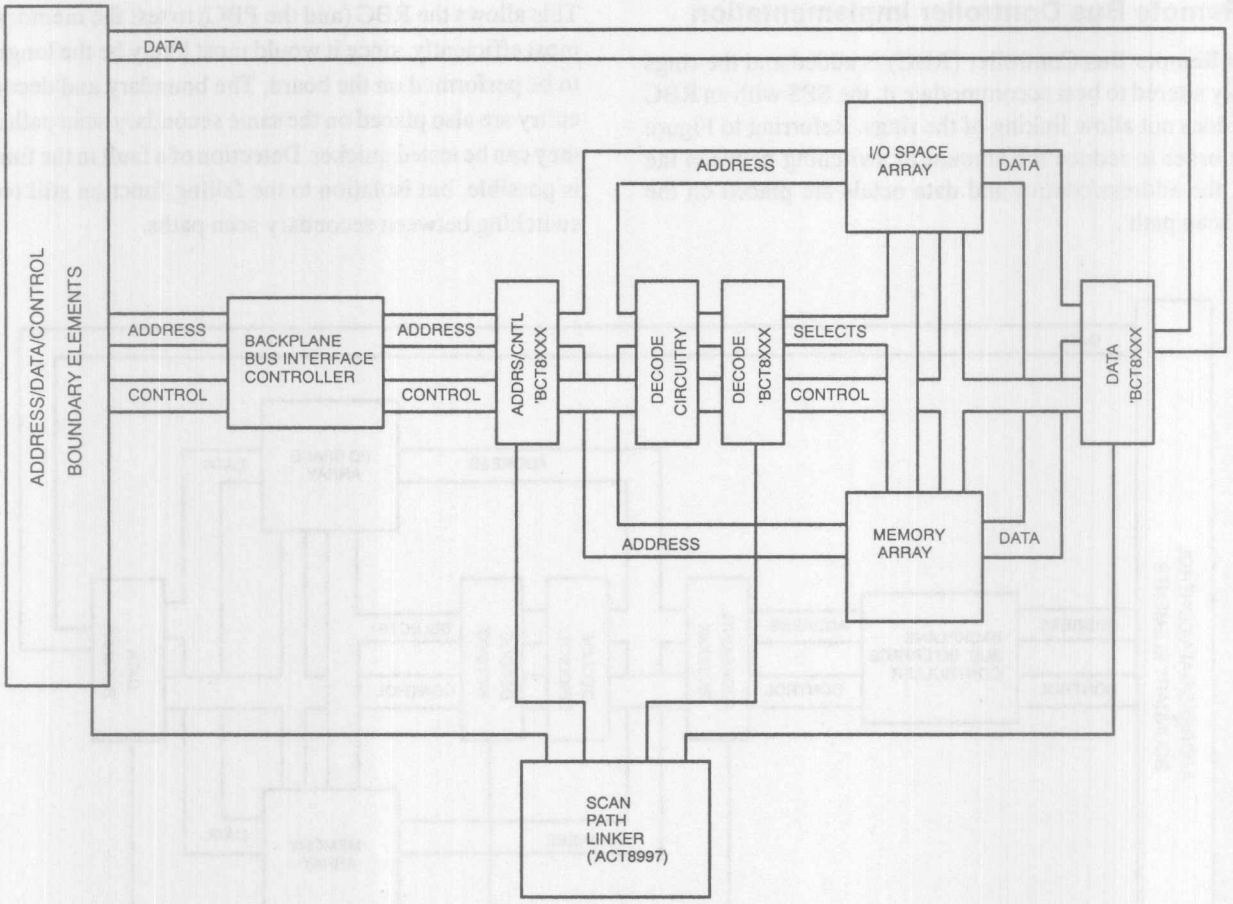


Figure 9. Memory Board Partition Example Using Scan-Path Linker

In addition, the backplane has been isolated by inserting the 1149.1-compatible octals for testing the interface to and from other boards using the test controller. These elements are on a scan path by themselves so that they can be scanned to perform the aforementioned tests without scanning the rest of the board. Next, 1149.1-compatible octals are placed between the bus controller and decode circuitry and between the decode circuitry and memory-I/O space arrays, each on its own scan path. Since tests of each of the functions relies on two of the scan-path device rings, the SPL is chosen. Thus, to test the memory array the control selects the address/control buffers and data buffers to be opened and links through the SPL; the boundary elements and address/control to test the bus interface controller; and so on. The address/control and decode octals could have been placed on the same scan path, but this would only serve to complicate the tests.

Remote Bus Controller Implementation

When Remote Bus Controller (RBC) is added and the rings slightly altered to best accommodate it, the SPS with an RBC input does not allow linking of the rings. Referring to Figure 10, in order to reduce the amount of switching between the rings, the address/control and data octals are placed on the same scan path .

This allows the RBC (and the PBC) to test the memory array most efficiently, since it would most likely be the longest test to be performed on the board. The boundary and decode circuitry are also placed on the same secondary scan path so that they can be tested quicker. Detection of a fault in the functions is possible, but isolation to the failing function still requires switching between secondary scan paths.

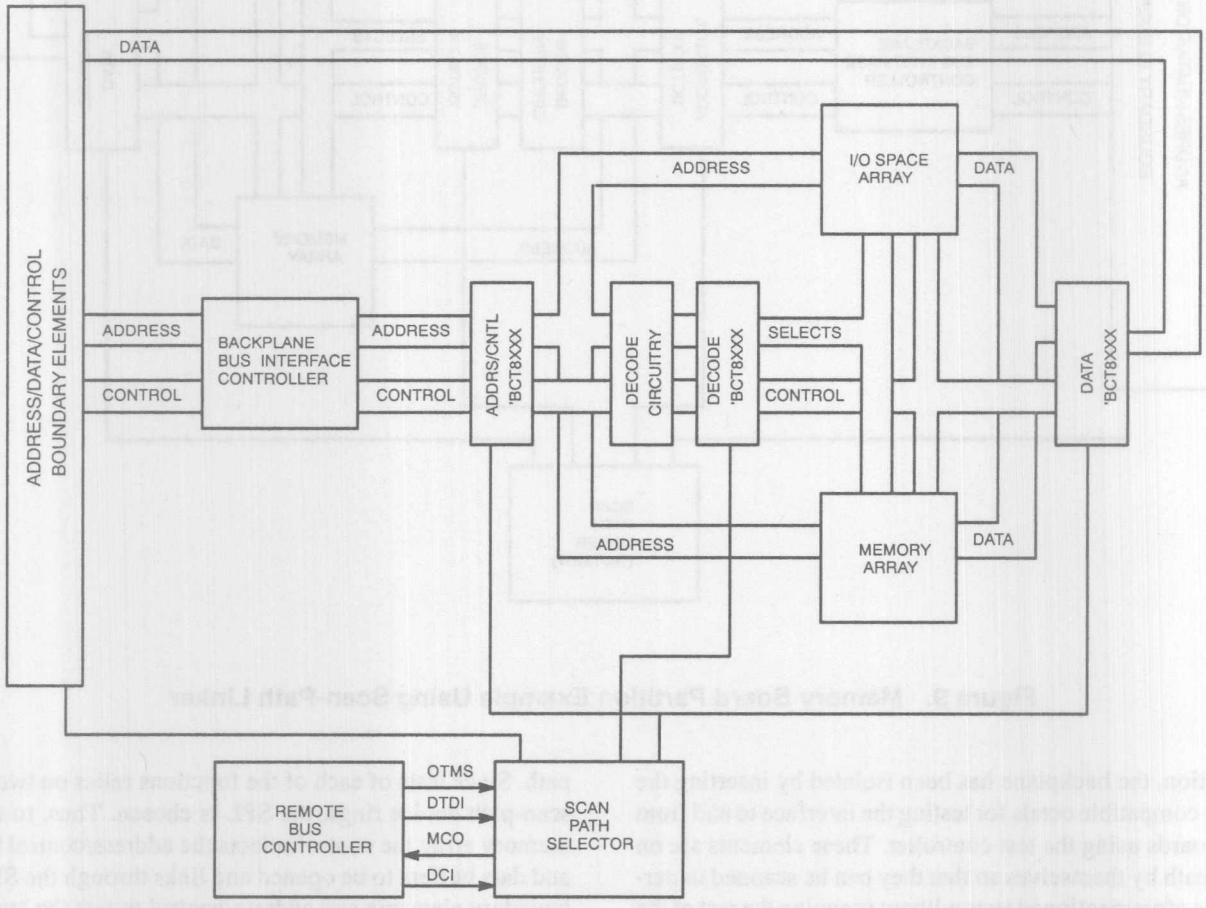


Figure 10. Memory Board Partition Example Using Scan-Path Selector

Prototype Testing Simplified by Scannable Buffers and Latches

by Andy Halliday, Greg Young, and Al Crouch

Abstract

Conventional logic devices incorporating boundary scan with the IEEE 1149.1 interface offer tremendous improvements in board testing. These benefits and improvements are contrasted against traditional approaches.

Introduction

The use of devices incorporating the IEEE 1149.1[1], hereafter referred to as 1149.1, boundary-scan architecture offers significant advantages when testing prototype systems. Prototype systems primarily are developed to allow engineers to prove a design concept or implementation before committing it to full production, where the cost to correct problems can be prohibitive. Engineers can verify such items as the basic design concept, theory of operation, board layout, parts selection, etc. Design flaws undetected during a paper design or simulation also can be corrected.

A prototype system is normally a low-cost, low-volume production of the end product. Most of the money spent during prototype development is devoted to building the prototype and verifying the design concept; very little usually is allocated for testing. Prototype testing usually is accomplished through simple functional operation and verification of the system as a whole. Functional tests of system hardware and software typically are performed on the entire system or pieces of the system, possibly with some unavailable functions being emulated externally. When failures are detected during this design verification process, a significant amount of time is spent determining the cause of the failure and correcting it. This debugging process is a manual, often time-consuming, task that does not ensure immediate success. Several hours may be spent rerunning the system after swapping boards and components that are believed to have caused the failure but, in fact, did not. Mixing design verification (concept, implementation, etc.) with fault verification can be costly in the long run if these processes do not complement one another.

Design verification and fault verification can be less painful by using devices that support the 1149.1 boundary-scan architecture and partitioning the system into small, easy-to-test functions. The use of boundary scan allows each partitioned function to be verified and tested independently, thereby reducing the time spent locating the cause of a failure. Boundary scan provides an increase in the controllability and observability of internal circuit nodes, which is mandatory when isolating system hardware faults and verifying operation of system software. Standard integrated circuits implementing the 1149.1 boundary-scan architecture are very beneficial in this situation. The use of such devices supports a hierarchical test philosophy in which the same test capabilities and test programs can be reused at each level (device, board, box, system).[2]

The following paragraphs compare traditional methods for design verification/debug/test of a prototype system with the method used for a prototype system containing boundary scan.

Traditional Test Methods

During prototype system design, testing issues are often far from the minds of the designers. If test is an issue, ad-hoc testability techniques sometimes are implemented. The design engineer primarily is concerned with how to implement a given piece of the system and have it interface correctly with the rest of the system. The software engineer is concerned with getting the system software working and verified on the system. The systems engineer is concerned with how the sys-

tem is going to function when all the pieces are put together. The complexity of the test problem comes to the surface when all the pieces are assembled and the system does not function properly. Determining why the system does not function as intended can be a major problem. Typical problems include:

- A manufacturing problem (miswire, wrong component, etc.)
- A design problem (incorrect design implementation)
- A software problem (incorrect algorithm)
- A hardware failure (bad part, etc.).

There are many approaches to identify the problem, but no real test strategy exists for debugging. The designer usually determines the process for obtaining a functional system.

The equipment used in the traditional test and debug process will vary depending on whether a whole system, a subsystem, or a few boards are being checked out. Types of equipment that are needed often include:

1. Hot mockup(s)
2. Hardware and software emulator(s)
3. Logic analyzer(s)
4. Oscilloscope(s)
5. Multimeter(s)
6. Specially designed debug box(es) (special test equipment)
7. Logic probe(s)

Depending on the design, the equipment cost may be very expensive and hard to justify for a low-volume system. User expertise and related training costs are other factors to consider.

A typical approach taken to verify/debug/test the hardware and software in a prototype system is:

1. Make a visual inspection of all the boards to check for any obvious problems; for example, wrong parts on the boards.
2. Do a continuity test of V_{CC} and GND to check for shorts. This always should be performed on each board before placing it in the system to avoid the possibility of damaging the whole system when power is applied.
3. At this point, there are many different options, depending on what the engineer decides is the best approach. One approach is:
   - Run the hardware and software, making necessary patches (either hardware or software), to get around parts not currently available. If everything runs as intended, it is assumed that no problem exists. If not, there are many ways to proceed including:
     - Lower the hardware complexity by removing boards and patching around the boards.
     - Lower the software complexity by changing the software
     - Execute the software in a single-step mode and attempt to identify the problem source by using a logic analyzer and/or an oscilloscope.

After verification of all the hardware and software for one complete system, this system becomes a "hot mockup" to be used for testing/debugging other boards received from manufacturing (or returned from field use). Using the hot mockup, other boards may be verified by:

1. Replacing the good board in the system with the Unit Under Test (UUT)
2. Running the hot mockup software
3. If the hot mockup software runs correctly, the UUT will be considered good. If not, the engineer(s) may attempt to isolate the problem by probing the UUT while the software is running.

The hot mockup approach is very iterative and may require significant time to be devoted to the debug/test effort. Additionally, swapping boards can induce faults into the system other than those faults associated with the unit under test.

Approaches other than the hot mockup include:

- The use of special tester boxes for testing boards, functions, etc.
- The use of software and hardware emulators
- The use of special test equipment.

The description above shows that traditional methods usually do not involve a structured design verification/testing strategy. Concurrent verification of hardware and software makes it difficult to isolate between hardware and software faults. The increasing complexity and density of today's systems may require costly equipment for verification and test, and board "real estate" must be allocated for probing.

Boundary-Scan Test Method

The poor fault isolation and lengthy, unstructured approach of traditional test methods suggest that a new method is needed. A hierarchical test methodology based on test capabilities

such as boundary scan can solve some of these problems. A hierarchical test concept benefits both the designer and the test engineer because system design verification (hardware and software) and testing are accomplished using the same methods. Such a concept stems from having test capabilities, such as boundary-scannable devices, incorporated into the design that can be used at all levels of test (device, board, box, system). Debug software and test programs using these capabilities are reusable for each level of equipment integration. This is possible by using a building-block approach to test software development and execution. Test routines developed to exercise certain equipment functions can be built upon to exercise additional equipment functions. The routines used in hardware debug can be built upon to create functional or pattern-oriented tests for the equipment.

System Description

To study the benefits of using standard devices incorporating 1149.1 boundary scan as a tool for hierarchical test, a prototype system was designed. This prototype system is shown in Figure 1.

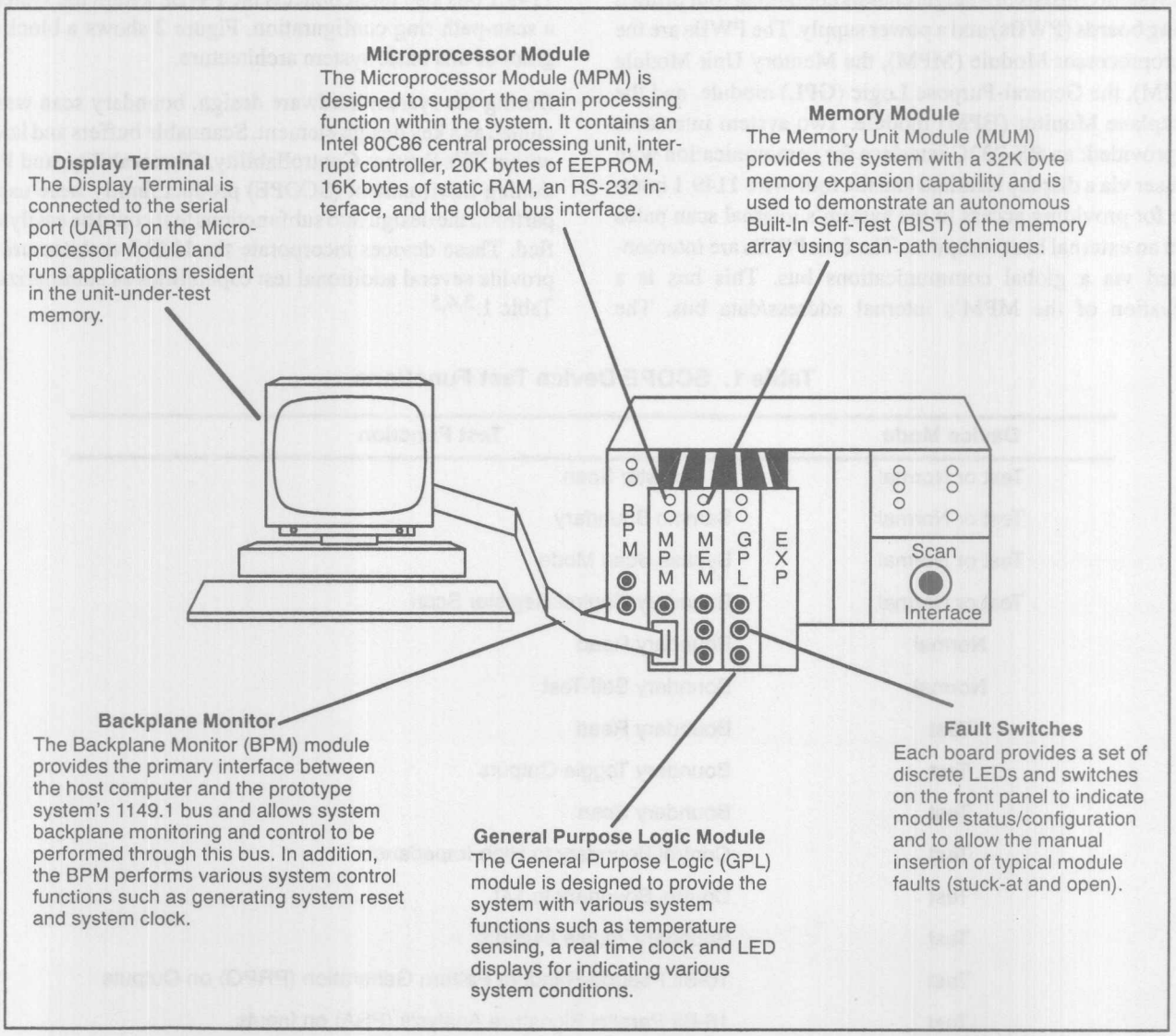


Figure 1. Prototype System

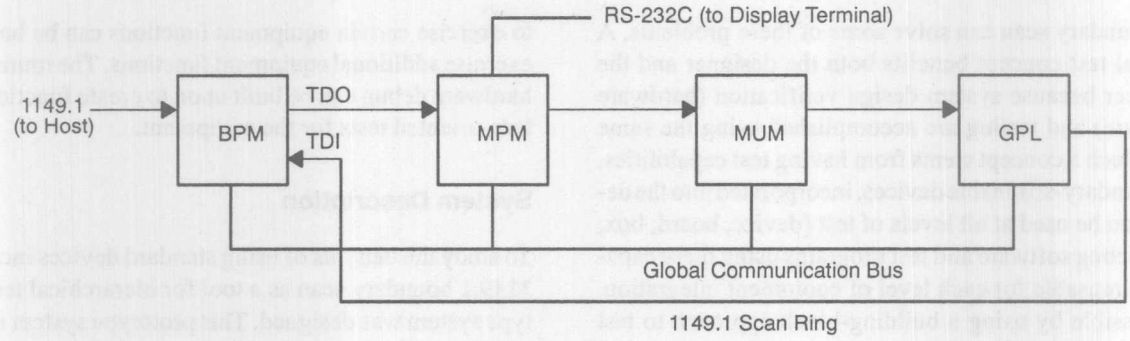


Figure 2. Prototype System Block Diagram

The system consists of a single chassis containing four printed wiring boards (PWBs) and a power supply. The PWBs are the Microprocessor Module (MPM), the Memory Unit Module (MUM), the General-Purpose Logic (GPL) module, and the Backplane Monitor (BPM) module. Two system interfaces are provided: an RS-232C interface for communication with the user via a display terminal and the four-wire 1149.1 interface for providing access to the system's internal scan paths from an external host computer. The four PWBs are interconnected via a global communications bus. This bus is a derivation of the MPM's internal address/data bus. The 1149.1 bus also interconnects the PWBs within the system in a scan-path ring configuration. Figure 2 shows a block diagram of this basic system architecture.

During the system hardware design, boundary scan was included as a key design element. Scannable buffers and latches within TI's System Controllability, Observability, and Partitioning Environment (SCOPE) product family were used to partition the design into subfunctions that could be easily verified. These devices incorporate the 1149.1 architecture and provide several additional test capabilities as summarized in Table 1.[3,4,5]

Table 1. SCOPE Device Test Functions

Device Mode	Test Function
Test or Normal	ID Register Scan
Test or Normal	Sample Boundary
Test or Normal	Bypass Scan Mode
Test or Normal	Boundary Control Register Scan
Normal	Boundary Read
Normal	Boundary Self-Test
Test	Boundary Read
Test	Boundary Toggle Outputs
Test	Boundary Scan
Test	Control Boundary to High-Impedance
Test	Control Boundary to 1/0
Test	Boundary Toggle Outputs
Test	16-Bit Pseudo-Random Pattern Generation (PRPG) on Outputs
Test	16-Bit Parallel Signature Analysis (PSA) on Inputs
Test	Simultaneous 8-Bit PSA and PRPG

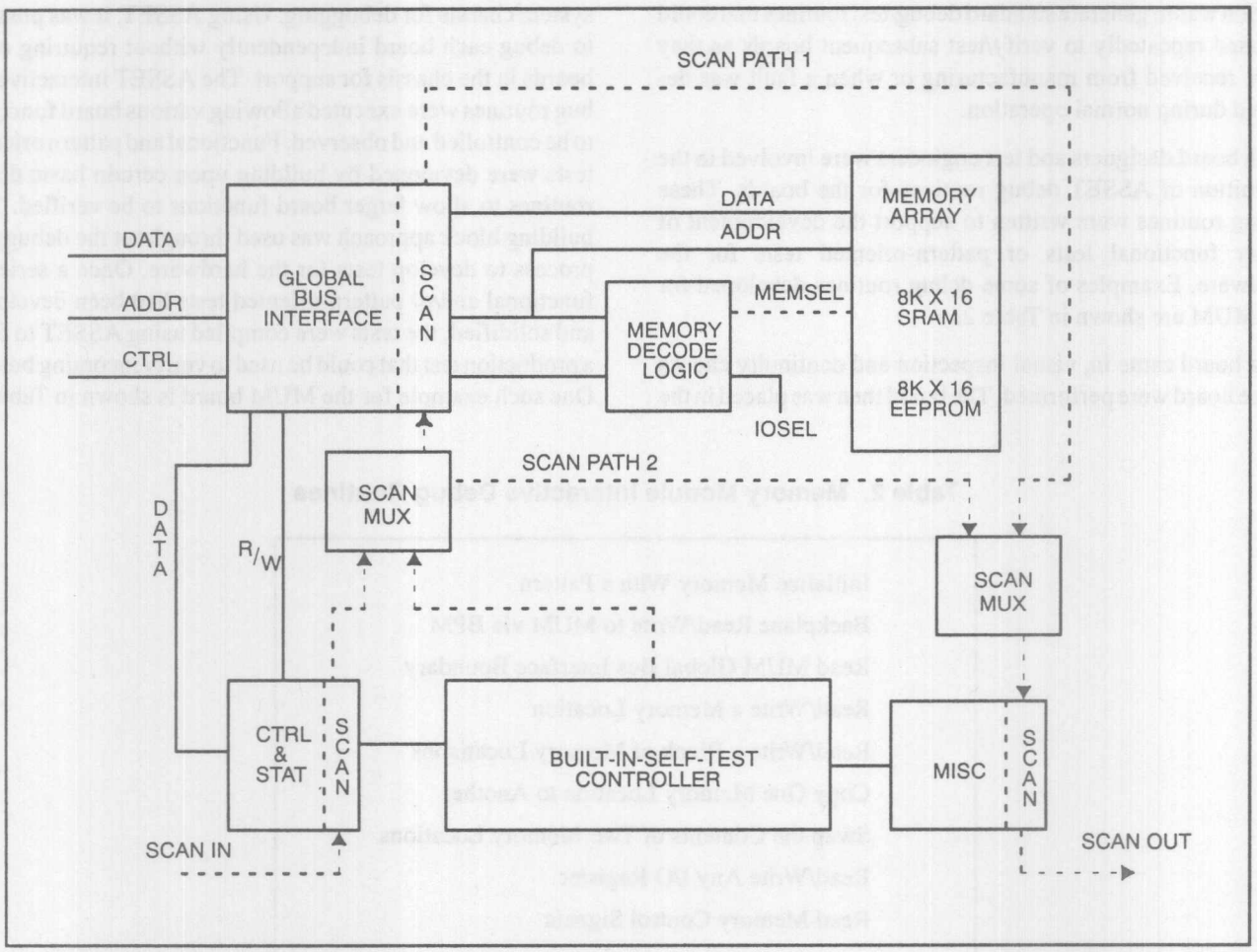


Figure 3. MUM Block Diagram

The SCOPE octal implementation in the MUM is shown in Figure 3. SCOPE octal devices are substituted for conventional octal devices such as buffers and latches and were placed strategically on data, address, and control signals. Thus, the MUM is divided into four distinct subfunctions, each individually accessible through the module's 1149.1 interface. These subfunctions are the global bus interface, the Built-In Self-Test (BIST) controller, the memory decoding logic, and the memory array [Static Random-Access Memory (SRAM) and Electrically Erasable Programmable Read-Only Memory (EEPROM)].

The target verification/test/debug environment consisted of TI's Advanced Support System for Emulation and Test (ASSET). ASSET is a test software development/execution environment hosted on an IBM PC-AT™ compatible computer. A Scan Controller Module (SCM) plugs into an expansion slot of the host computer and provides the ASSET software with an interface to the target system's 1149.1 bus. ASSET software is an extension of the C programming language to support boundary scan. In addition, ASSET software provides a library of low-level functions for controlling 1149.1-based boundary-scan devices and an interactive debugging utility that allows the user to manipulate each of those devices. The library functions are used to control the SCOPE device test capabilities described in Table 1.[6]

Methodology Employed

The goal of the prototype system development was to demonstrate the test and debug capabilities provided by the SCOPE octals when supported in the ASSET test environment. Each board design engineer generated their own board debug routines using ASSET and use the boundary-scan interface as much as possible during board debug. Traditional test equipment was to be limited to an ohmmeter for performing continuity checks before applying power to the board. The in-

tention was to generate standard debug/test routines that could be used repeatedly to verify/test subsequent boards as they were received from manufacturing or when a fault was detected during normal operation.

Both board designers and test engineers were involved in the definition of ASSET debug routines for the boards. These debug routines were written to support the development of either functional tests or pattern-oriented tests for the hardware. Examples of some debug routines developed for the MUM are shown in Table 2.

As a board came in, visual inspection and continuity checks of the board were performed. The board then was placed in the system chassis for debugging. Using ASSET, it was possible to debug each board independently without requiring other boards in the chassis for support. The ASSET interactive debug routines were executed allowing various board functions to be controlled and observed. Functional and pattern oriented tests were developed by building upon certain basic debug routines to allow larger board functions to be verified. This building block approach was used throughout the debugging process to develop tests for the hardware. Once a series of functional and/or pattern-oriented tests had been developed and solidified, the tests were compiled using ASSET to form a production test that could be used to verify incoming boards. One such example for the MUM board is shown in Table 3.

Table 2. Memory Module Interactive Debug Routines

Initialize Memory With a Pattern
Backplane Read/Write to MUM via BPM
Read MUM Global Bus Interface Boundary
Read/Write a Memory Location
Read/Write a Block of Memory Locatiaons
Copy One Memory Location to Another
Swap the Contents of Two Memory Locations
Read/Write Any I/O Register
Read Memory Control Signals

Table 3. Sequence of Memory Module Routines Used to Form Board-Level Test

Scan-Path Continuity Test
BPM to MUM I/O Continuity Test
MUM to BPM I/O Continuity Test
Board Identification Test
Chip Select/Decode Logic Test
Memory Tests
Address Uniqueness Test
March II Algorithm Test

Lessons Learned

Through development of the prototype system, we were able to discover the major benefits of using boundary scan versus using traditional methods for design verification, debug, and test. We also learned several design and test practices that were beneficial to our effort. Most of the lessons learned fall into one of the following categories: fault isolation, design partitioning and test access, ease of use, scan-path design, and structured process. The following paragraphs discuss these categories and specific lessons, design rules, and observations that were made.

- Fault Isolation (Traditional Versus Boundary Scan)

  As the first set of boards was being debugged, ASSET was still under development. Code syntax and format changes, along with compiler changes, forced traditional methods to be used in some cases. Because of this, the MPM designer decided to use traditional methods in debugging his board, which slowed the effort considerably. While using traditional methods, the designer was unable to isolate a code execution problem. He eventually resorted to using ASSET after several days of tracking the problem without success. Using the boundary scan within the module, the memory was loaded and verified both directly (without using decode logic) and indirectly (using decode logic). This allowed the problem to be isolated to a design error within the memory decode logic.

  Another example of how ASSET and boundary scan were successful in isolating various system problems was in debugging of the MUM board. A miswired bus-enable signal was creating bus contention on data lines. ASSET helped isolate the problem quickly, without requiring any manual probing of the board.

  In addition to the design errors that were isolated, several component and signal failures also were isolated. Switches purposely were designed into each module to allow stuck-at and open faults to be simulated. The types of faults easily isolatable through the boundary-scan method included backplane faults, memory decode faults, internal bus faults, and scan-path faults. These types of faults are typically very difficult to isolate with traditional methods. The scan-path fault was isolated through the use of the preload feature built into scan instruction register command of the SCOPE octals, as defined by 1149.1.

- Design Partitioning and Test Access

  Experimentation using the fault insertion switches reinforced some important guidelines regarding boundary-scan placement and design partitioning. It is very important that boundary scan be implemented at functional partitions, especially the board-to-board interface, to allow functions to be exercised and isolated completely. Also, boundary scan access to key control signals, such as the microprocessor HOLD signal, is important to avoid bus contention when attempting to drive buses with boundary-scan registers.

  By providing controllability and observability of each board's backplane signals through boundary scannable devices, each board could be tested independently as it arrived from manufacturing. Emulation of board interfaces, whenever required, was controlled through the backplane boundary scan. Using this method, board-to-board dependencies were eliminated, and each could be verified standalone. This was very beneficial during the integration phase, since the system could be debugged without requiring all boards in place, thereby reducing the ambiguity size whenever a problem was detected. The MUM and GPL boards were verified completely using ASSET and the boundary-scan interface. Once two boards were debugged, a system backplane test then could be performed between boards using ASSET. This allowed the backplane interconnects between two boards to be verified using a pattern-specific test, as opposed to implicitly testing them through interaction between the MPM and either of these modules. In this way, faults caused by the backplane wiring or connector seating were detected and isolated easily.

- Ease of Use (ASSET)

  Using ASSET to debug/test each board proved to be less time-consuming than taking the traditional approach. For example, the designer of the GPL was able to write the ASSET debug routines and complete debug of his board in a few days with no experience in writing ASSET code or using the system. Once the GPL tests had been compiled into a production test for the board, the remaining GPL boards were verified quickly, without resorting to any of the traditional methods.

  Both test engineers and software engineers also were involved in the use of ASSET to debug the system and create production tests.

- Scan-Path Design

  While there are obvious benefits to using boundary scan within a design, there are also problems that can be created if it is implemented incorrectly. The instances in which this occurred during the design/development of the prototype system dealt with scan clock control and changing of scan-

path lengths. These problems can be eliminated through careful adherence to scan-path design rules.

Some BIST designs, such as the one implemented on the MUM, may require explicit clock control over independent scan paths and, therefore, mandate that scan clocks be gated. Careful design should be used when such gating is necessary. During debugging of the MUM, this gating caused several problems when attempting to use scan paths under control of the MUM BIST. When inactive, the BIST disabled certain scan clocks, and all scan data beyond this point were made inaccessible. The problem would have required manual probing to isolate had it not been for the 1149.1 instruction register scan preload of data. When commanded to do an instruction register scan, the SCOPE octals preload a binary 10 on the two least-significant bits of the scan data output.[7] Those octals' scan paths that are obstructed in some way, however, will output all zeros (0s) or all ones (1s), depending on the fault. By analyzing the information retrieved during a single instruction register scan, the point at which the scan data were corrupted can be determined.

While the 1149.1 specification allows scan clocks to be disabled, it does not encompass the system design considerations necessary to ensure the operation of a device's additional test capabilities, such as those available in the SCOPE octals. A system design problem of this type was discovered when developing backplane tests that use the SCOPE PRPG/PSA modes. The BPM backplane scan path is implemented as a separate scan ring from the other PWBs within the system (MPM, MUM, GPL). The two separate rings are connected in parallel with each interfacing to the host through the same scan-path signals. Because of this architecture, circuitry was added to ensure that only one of the rings could be active (scan clock enabled) at any given time. As a result, backplane PRPG/PSA tests between modules on separate scan rings could not be performed; for example, a test between the BPM and MPM could not be performed. This forced the development of two separate backplane tests: one using the BPM (discrete patterns) and the other using only PWBs on the secondary ring (free-running PRPG/PSA). To avoid this problem, the same free-running clock must be provided to each ring if backplane tests using SCOPE PRPG/PSA modes are to be performed between boards on different rings.

Another problem encountered concerns the collapsing and expanding of scan rings within a design. For example, while designing the MUM BIST, it was found that, while collapsing and expanding of scan rings is acceptable, it should not be performed arbitrarily with respect to the test bus controller. Changing the scan-path length without notifying the test bus controller will cause the controlling software to become confused and possibly not recover. All scan-path length changes should be done in conjunction with the test bus controller to be as graceful as possible.

- Structured Process

  One of the most important discoveries during development of the prototype system was the fact that the use of ASSET and boundary scan imposed a structured test and debug process for the engineer. As mentioned earlier, a great portion of the MPM was debugged using lengthy traditional methods. The process did not benefit the debug/test of the second and third MPMs when they arrived. Since the design engineer chose not to write ASSET debug routines, even after ASSET had become stable, only low-level ASSET library functions were available to manipulate the board's boundary-scan devices. Because the majority of the time was spent using traditional debug methods, no ASSET debug routines or MPM production test could be developed within the given timeframe.

  In contrast, the engineers who used ASSET for design verification/debug/test of the MUM and GPL followed a structured procedure that was repeatable on subsequent boards. This procedure uses a building-block approach whereby individual pieces of the system are verified and then may be used to verify the remaining pieces. The procedure is shown in Figure 4 and can be divided into the following four phases:

  - Hardware design
  - Debug/test design
  - Debug/test development and execution
  - Test compilation

  The hardware design phase consists of implementing boundary-scan devices into the design to partition it into easily testable functions.

  During the debug/test design phase, a debug strategy for each board (and subfunctions) is defined and the low-level ASSET functions required to implement the debug routines are identified.

After the debug routines are defined, ASSET software may be written and executed on the designs. The debug routines may be combined to form test routines that, in turn, can be combined to form the board-level production test.

Once a production test exists for each board, they may be combined with a system backplane test to form a production test for the system.

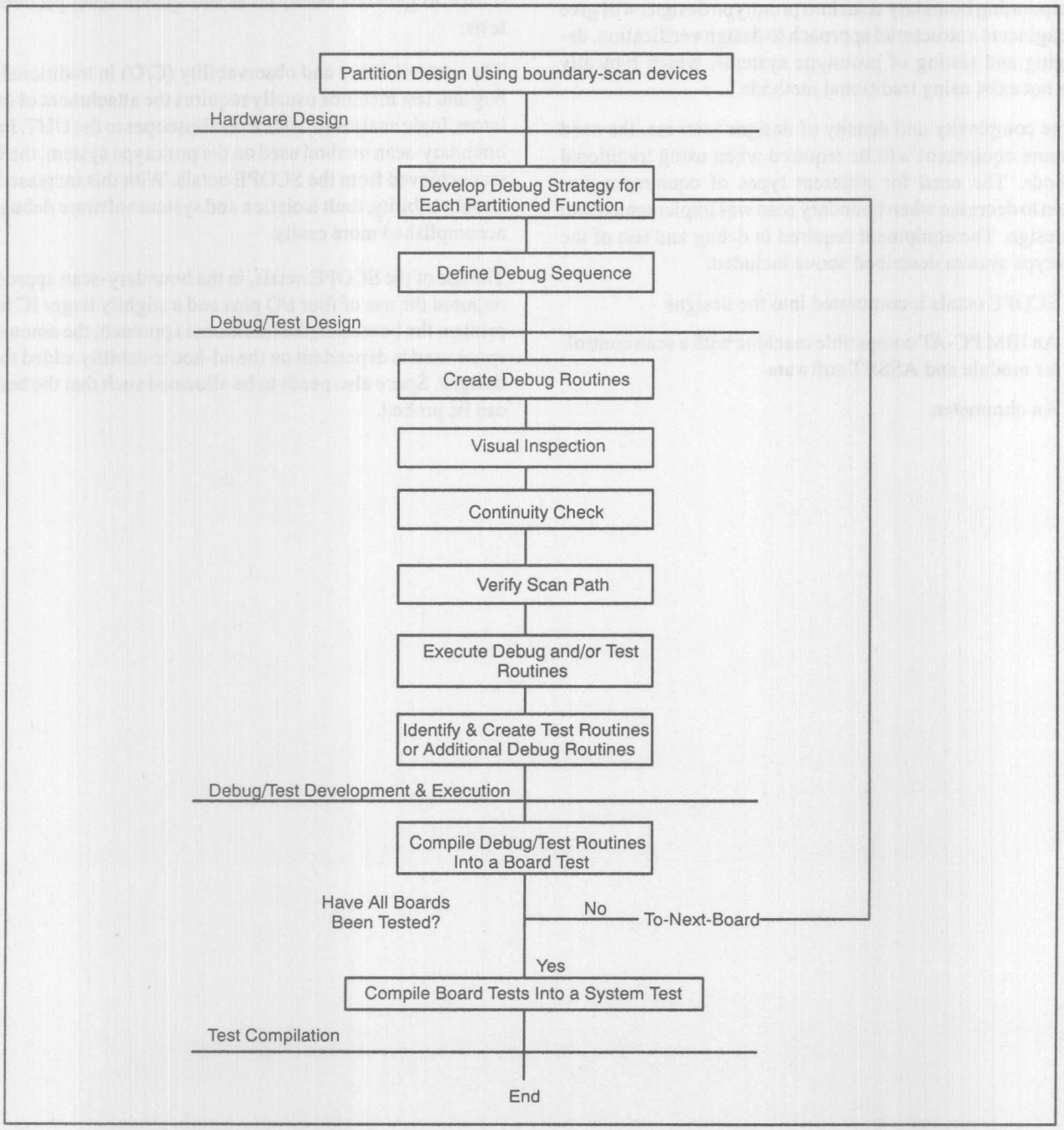


Figure 4. Structured Debug/Test Procedure

Benefits of Boundary-Scan Methods Versus Traditional Methods

The key benefits of boundary-scan methods versus traditional methods for design verification/debug/test are shown in Table 4 and discussed in the following paragraphs.

Incorporating boundary scan into prototype designs will give the engineers a structured approach to design verification, debugging and testing of prototype systems, which typically does not exist using traditional methods.

As the complexity and density of designs increase, the need for more equipment will be required when using traditional methods. The need for different types of equipment was shown to decrease when boundary scan was implemented into the design. The equipment required in debug and test of the prototype system described above included:

1. SCOPE octals incorporated into the designs
2. An IBM PC-AT compatible machine with a scan controller module and ASSET software
3. An ohmmeter.

In traditional methods, the software generated by engineers during design verification and debug usually is discarded after verification of the system. If the software is not discarded, it rarely can be used for board and system testing. In contrast, the hierarchical test method used on the prototype system allowed the software generated for board-level debug to be reused to generate board-level and system-level production tests.

The controllability and observability (C/O) in traditional debug and test methods usually requires the attachment of emulators, logic analyzers, and/or oscilloscopes to the UUT. In the boundary-scan method used on the prototype system, the C/O was achieved from the SCOPE octals. With this increased internal visibility, fault isolation and system software debug are accomplished more easily.

The use of the SCOPE octals, in the boundary-scan approach, required the use of four I/O pins and a slightly larger IC footprint on the boards. In the traditional approach, the amount of space used is dependent on the ad-hoc testability added to the designs. Space also needs to be allocated such that the boards can be probed.

Table 4. Differences in Traditional and Boundary-Scan Methods

Tradeoffs	Traditional	Boundary Scan
Approaches	Usually not structured	Structured approach
	Usually starts after design has gone to manufacturing	Starts during design
Test Equipment	Complexity/density of designs require more test equipment to test and debug systems	Limited equipment needed to test and debug systems. No need for expensive special test equipment during production test, depot test, etc...
	All "Hands On"	"Hands On" limited
	Hot mockup, logic analyzer, oscilloscope, ...	PC-AT software, ohmmeter test octals
Software	Tests generally not reusable	Reusable test software go from debug—board—production ...
Internal Visibility	Fault isolation depends on ad-hoc testability	Fault isolation increased
	Manual probing required	No manual probing required
	Required external hardware to look at internal nodes	Have internal node visibility
Real Estate	Additional real estate required to support ad-hoc testability and space to allow for probing	Additional real estate, four I/O pins, and a larger footprint IC

Conclusion

The increasing complexity of designs, the use of double-sided, surface-mounted boards, the spacing of components decreasing, etc., will warrant a new design verification and test approach. Devices incorporating boundary scan impose a minimal real estate overhead and change the process of design verification and test, which make it beneficial to both the design engineer and test engineer. The use of devices incorporating boundary scan will reduce the cost of test.

By using the devices that support the 1149.1 architecture in our prototype system, some of the problems and questions associated with the verification and testing of prototype systems (or even production systems) were solved. In addition to solving the problems, the verification and test process was simplified.

References

[1] *Proposed IEEE 1149.1, Standard Test Access Port and Boundary-Scan Architecture*, Draft D3, January 18, 1989.

[2] Pete Fleming. *An Advanced Test Architecture—What's in it for You?,* Texas Instruments Technical Journal, July-August 1988, pp. 17–27.

[3] Lee Whetsel. *A Standard Test Bus and boundary-scan architecture,"* Texas Instruments Technical Journal, July-August 1988, pp. 48–59.

[4] Lee Whetsel. *Standard Test Port and Cells Provide An ASIC Testability Toolkit,* Texas Instruments Technical Journal, July-August 1988, pp. 35–43.

[5] Lee Whetsel. *A Proposed Standard Test Bus and boundary-scan architecture*, Proceeding of the International Conference on Computer Design, October 3–5, 1989, pp. 330–333.

[6] Don McClean. *Making the Overhead of Scan Transparent (Almost!)*, Texas Instruments Technical Journal, July-August 1988, pp. 145–151.

[7] Don McClean and Javier Romeu. *Design for Testability With JTAG Test Methods*, Electronic Design, June 8, 1989.

[8] *Joint Test Action Group Specification*, Version 2.0.

PSA-PRPG Techniques with SCOPE

by Adam Cron

Abstract

This application note describes the use of Texas Instruments ASIC Library SCOPE Cells in the construction of built-in-test structures that generate a pseudo-random pattern of numbers, or develop a parallel signature of data inputs. These PRPG/PSA test registers are accessed via the IEEE 1149.1 scan interface of an ASIC, and are compatible with the IEEE 1149.1 standard.

Objective

This application note is intended to give a design engineer a basic overview of how to implement Parallel Signature Analysis (PSA) and Pseudo-Random Pattern Generation (PRPG) structures for testability using Texas Instruments ASIC Library SCOPE components. It assumes that the designer has an understanding of PSA and PRPG techniques.

Introduction

The register used for PSA and PRPG techniques is called a Linear Feedback Shift Register (LFSR). This implementation is designed with test logic compatible with the IEEE 1149.1 specification. Several support circuits are also presented and elaborated upon.

What Are PRPG and PSA?

PRPG

PRPG is a method whereby data is driven from a test register to the logic block to be tested. This register is configured as a LFSR, and is preloaded with a starting, or seed, value before beginning the test. During the test, patterns are generated from the test register with each successive clock (and in the case of IEEE 1149.1-compatible test registers, on the falling edge of TCK).

PSA

PSA is used to compress the outputs of a logic block into a test register such that a unique value is obtained in the test register after a PSA test is run. Like PRPG, a PSA test register is configured as a LFSR and preloaded with a seed value. On each successive clock to the system and test register, a resulting new signature value is calculated and stored in the test register. This signature is based on the LFSR's current value and the test register inputs (upon receiving the rising edge of TCK in the case of IEEE 1149.1-compatible implementations).

PRPG and PSA

Together, these techniques could be used concurrently in a design to test logic blocks in that design. (Refer to Figure 1 for an example illustration.) PRPG patterns could be output from a PRPG test register (on the falling edge of a clock, for instance) while a signature is developed by a PSA test register (on the rising edge of the clock, for instance) based on the outputs of the logic block. The PSA/PRPG tests occur in the Run-Test/Idle state of the Test Access Port.

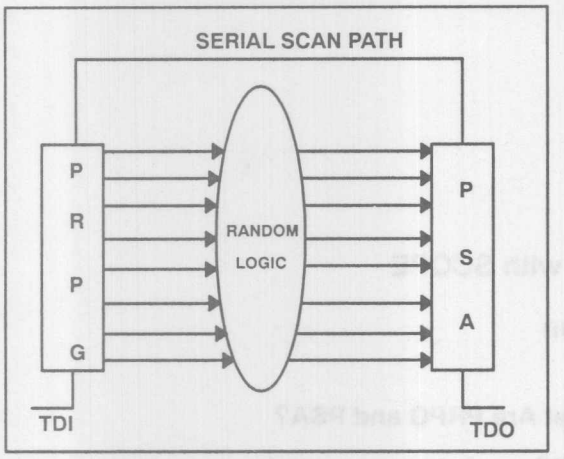


Figure 1. Testing with PRPG and PSA

Access to PSA/PRPG Test Registers

The PSA and PRPG test registers should be accessible via the IEEE 1149.1 scan interface to the ASIC using these registers. The scan bus is used to load the test registers with the seed value. Then the test clock (or system clock for some applications) clocks the test registers to generate the test patterns and signature. Then, via the scan bus, the signature in the PSA test register are scanned out and compared by the test host with a known good value (perhaps obtained from simulating the ASIC or system). In the general case, access does not have to be provided to the IEEE 1149.1 bus. However, to diagnose a failure based on these testing techniques, access to the test register improves the test and diagnostic process.

Benefits

1. The test host need only store seed values for the PSA and PRPG test registers, the correct signature value, and perhaps the number of clocks necessary to run the test.
2. Exhaustive test vectors do not have to be stored as they do for normal boundary-scan testing techniques.
3. The test time needed to test an ASIC in a system will usually be drastically reduced due to the fact that individual test patterns are not being scanned to the target (through other ICs), but are being generated by the test logic itself, after only a few scan operations. The test can then proceed at the test clock rate, with patterns and signatures being generated in each clock cycle.

Penalties

1. Because no actual functional test patterns are used to test the logic block, only a "go/no-go" test indication can be determined. If the PSA/PRPG test fails, a more exhaustive diagnosis of the failed logic blocks must be performed. The SCOPE architecture and the implementation presented in this application note should allow the test registers to perform both the PSA/PRPG "go/no-go" test and the exhaustive test.
2. Each design using this test technique may require a different number of clocks to complete the test. In fact, the IEEE 1149.1 specification requires that an ASIC running BIST must be able to complete the test using a free-running test clock. For this reason, a method internal to the ASIC may have to be provided to halt the test after the required number of clock cycles has been reached.
3. Due to the increased complexity of the LFSR structures and testability support functions, more silicon area will be taken up by the test structures in the ASIC.
4. Because this test is based on a compressed data format (the signature), it is possible that a) a fault in the logic could be masked by another fault, or b) a bad signature could map to two or more fault sources.
5. Reduced performance may be a by-product of the added testability logic.

PSA/PRPG Test Register Architecture Implementation

(Through-out the following discussion, you may find it helpful to refer to Figure 7.)

Figure 2 illustrates a four bit LFSR structure using the SCOPE cells. The control input signals are fully programmable, or generic, and thus, this register could be used for PSA, PRPG, or both at one time or another.

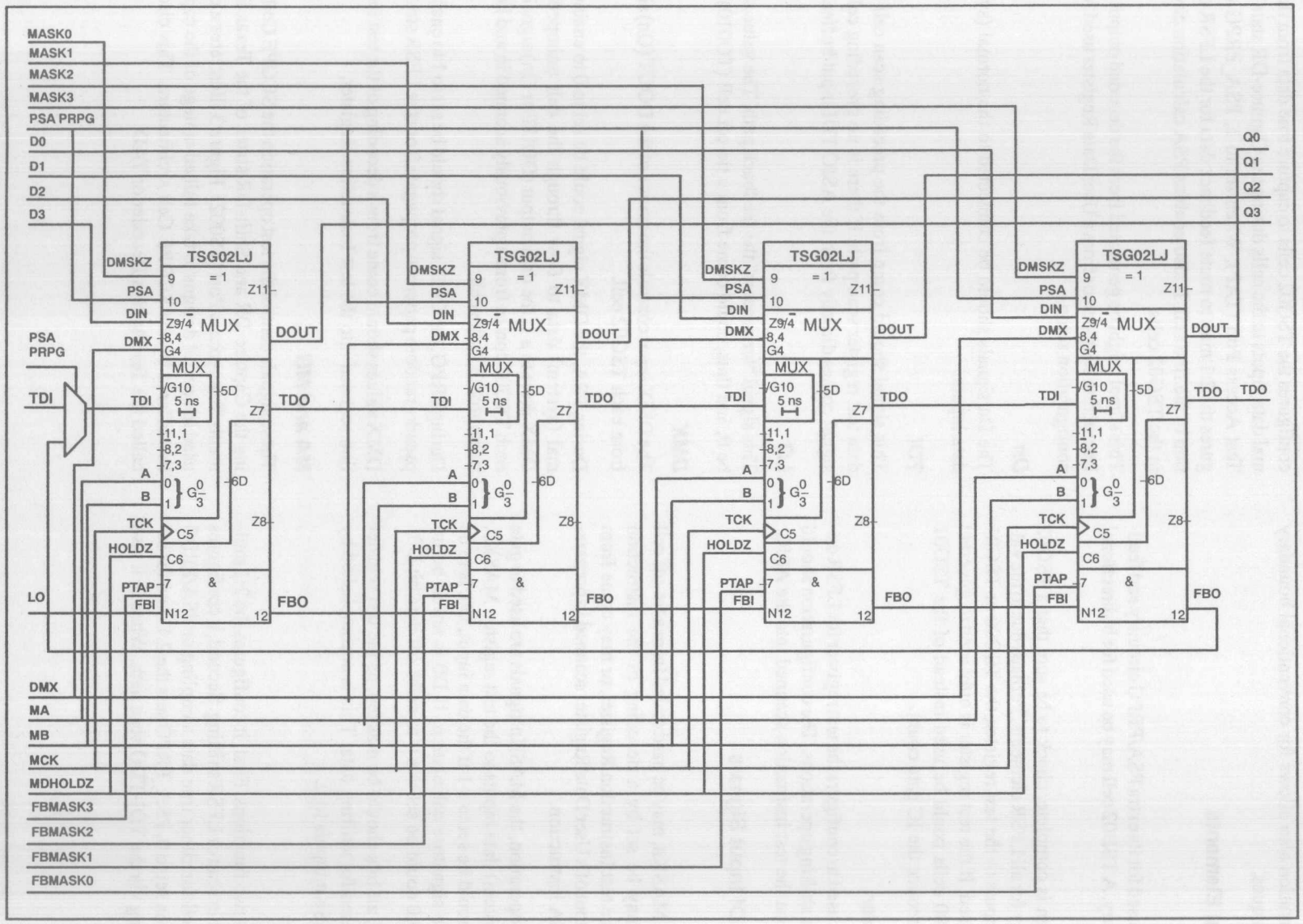


Figure 2. PSA/PRPG Register Built with SCOPE ASIC Cells

This implementation also allows for conventional boundary scan test techniques.

Architectural Elements

TSG02

The TSG02 is used for its extra PSA/PRPG circuitry and feedback tap circuitry. A TSB02 cell may be used for bidirectional paths.

After the design is complete, check to be sure that a TSG02 cell is necessary for all LFSR elements. Perhaps due to the value of certain inputs to the test register, the TSG00 or TSG01 cell could be used. If the test register is only used to generate patterns, TSG00 cells could be used instead of the TSG01 cells and thus reduce the IC gate count.

2:1 Multiplexer

The 2:1 mux is used to configure the test register for LFSR operations or data shifting operations. The configuration should depend solely on the test instruction scanned into the ASIC.

Description Of Input Signals

MASKn

The mask bits, MASKn, may be hard coded from a tie-off cell (TO010), or may be set by a decoding of the instruction scanned into the test Instruction Register, or may come from the parallel output of a User Data Register scanned in preparation for the PSA instruction.

During a PSA operation, the MASKn signals are used to gate the normal (system) data inputs to the test register. MASK0, for example should be set to a 1 if the data input, D0, is to be included in the signature calculation. If D0 is not to be included, MASK0 could be set to a 0 to mask off data bit D0.

During PRPG, all bits should be masked, as the test register itself is only used for shifting data. This is accomplished by setting all MASKn inputs to 0.

PSA_PRPG

This signal has two functions. First, it configures the 2:1 multiplexer for either scan or LFSR shifting. Second, it configures each TSG02 cell for either true data sampling or PSA/PRPG operation. When set to 0, PSA_PRPG sets the 2:1 mux for serial data shifting via the TDI-TDO scan path. When 0, it also configures the TSG02 cells to capture true data from the normal data inputs to the cells during the Capture-DR state of the Test Access Port (TAP). When set to 1, PSA_PRPG configures the 2:1 mux to route feedback data for the LFSR operation to the first cell. It also enables PSA calculation circuitry in the TSG02 cells.

This signal might be generated from the decoded output of the Instruction Register, or from a User Data Register used for test configuration setup.

Dn

The Dn signals should be connected to the normal (system) data inputs.

TDI

This signal should come from the preceding scan cell in the data test register scan path. If there is no preceding cell, this signal comes directly from the ASIC TDI input buffer.

LO

This signal "terminates" the feedback path. The value should be 0, and thus, could come from a tie-off cell (TO010).

DMX

The DMX input controls the source of the DOUT (Qn) signals from each TSG02 cell.

During PSA, the DMX signal could be set to 0 to enable normal (system) data to flow through the cell unimpeded. If DMX is set to a 1, the data output (DOUT or Qn) signal from each TSG02 comes from a previously scanned in and latched value stored in the cell.

During PRPG, the DMX signal should be set to 1 to enable the pseudo-random pattern to propagate from the LFSR structure.

DMX values should come from a decoding of the test instruction scanned into the test Instruction Register.

MA and MB

These signals control the test operation the SCOPE Cells during the Capture-DR and Shift-DR states of the Texas Instruments Test Access Port (TS002). Figure 3 illustrates a circuit used in several designs to take full advantage of the capabilities offered by the SCOPE Cell architecture. This circuit is called the Test Access Decoder or TAD.

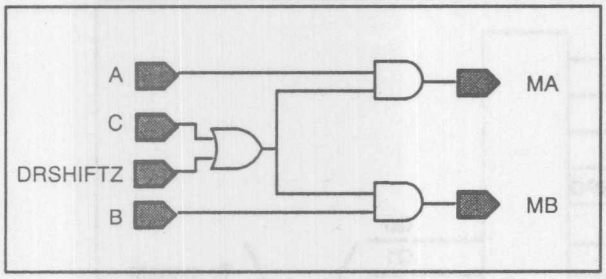


Figure 3. Test Access Decoder

A, B, and C come from a decoding of the test Instruction Register. Signal A and B control the function of the SCOPE Cell during the Capture-DR state of the TS002. C (in conjunction with A and B) controls the operation of the SCOPE Cells during the Shift-DR state of the TS002. The DRSHIFTZ signal comes from the TS002 TAP controller.

The diagrams in Figure 4 illustrate the function of the SCOPE Cells for the two logic values of C.

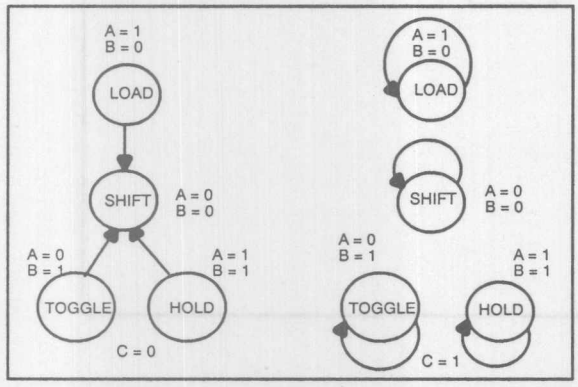


Figure 4. Actions Taken in Capture-DR and Shift-DR States of the Test Access Port

For PSA and PRPG operations, A, B, and C should be set to 1, 0, and 1, respectively. This configures the SCOPE Cells to always load (capture).

MCK

The source of MCK changes with the PSA_PRPG signal. During data shift operations, MCK should be provided by DRCK from the TS002. During LFSR operations, MCK should come from TCK while the TS002 is in the Idle/Run Test state. The circuit in Figure 5 shows one implementation of this function.

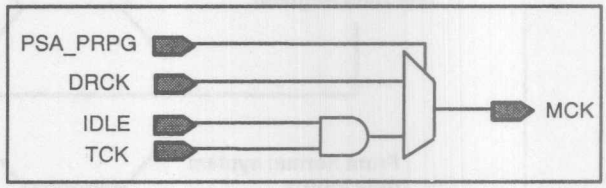


Figure 5. Test Clock Selection and Gating

The IDLE signal is an output of the TS002. It is active (high) when the TS002 is in the Idle/Run-Test state.

MDRHOLDZ

For PSA operations in which the DMX signal is 1, MDRHOLDZ should be held to 0 to keep a stable value propagating from the DOUT (Qn) signals of the TSG02 cells.

For PRPG operations, MDRHOLDZ should follow the DRHOLDZ signal from the TS002. However, in order to satisfy the IEEE 1149.1 specification, action from outputs of the test register need to occur on the falling edge of TCK. This may be accomplished by gating DRHOLDZ with TCK as shown in Figure 6.

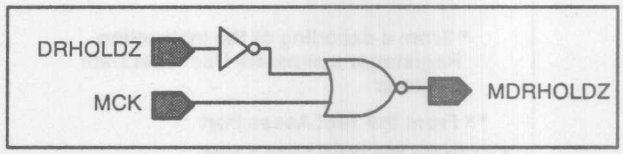


Figure 6. Circuit for Use with PRPG Operations

Selection of the correct MDRHOLDZ signal depends on the instruction scanned into the Instruction Register.

FBMASKn

To select a specific TSG02 cell as a provider of feedback data for a specific LFSR application, set the corresponding cell's FBMASKn input signal to 1. To disable the cell from providing feedback data, set the FBMASKn signal to 0.

The FBMASKn signals could come from a tie-off cell (TO010), from a decoding of the test instruction, or from a User Data Register providing this data.

Note that if a FBMASKn signal is always 0, a TSG01 cell could be used instead of a TSG02 cell.

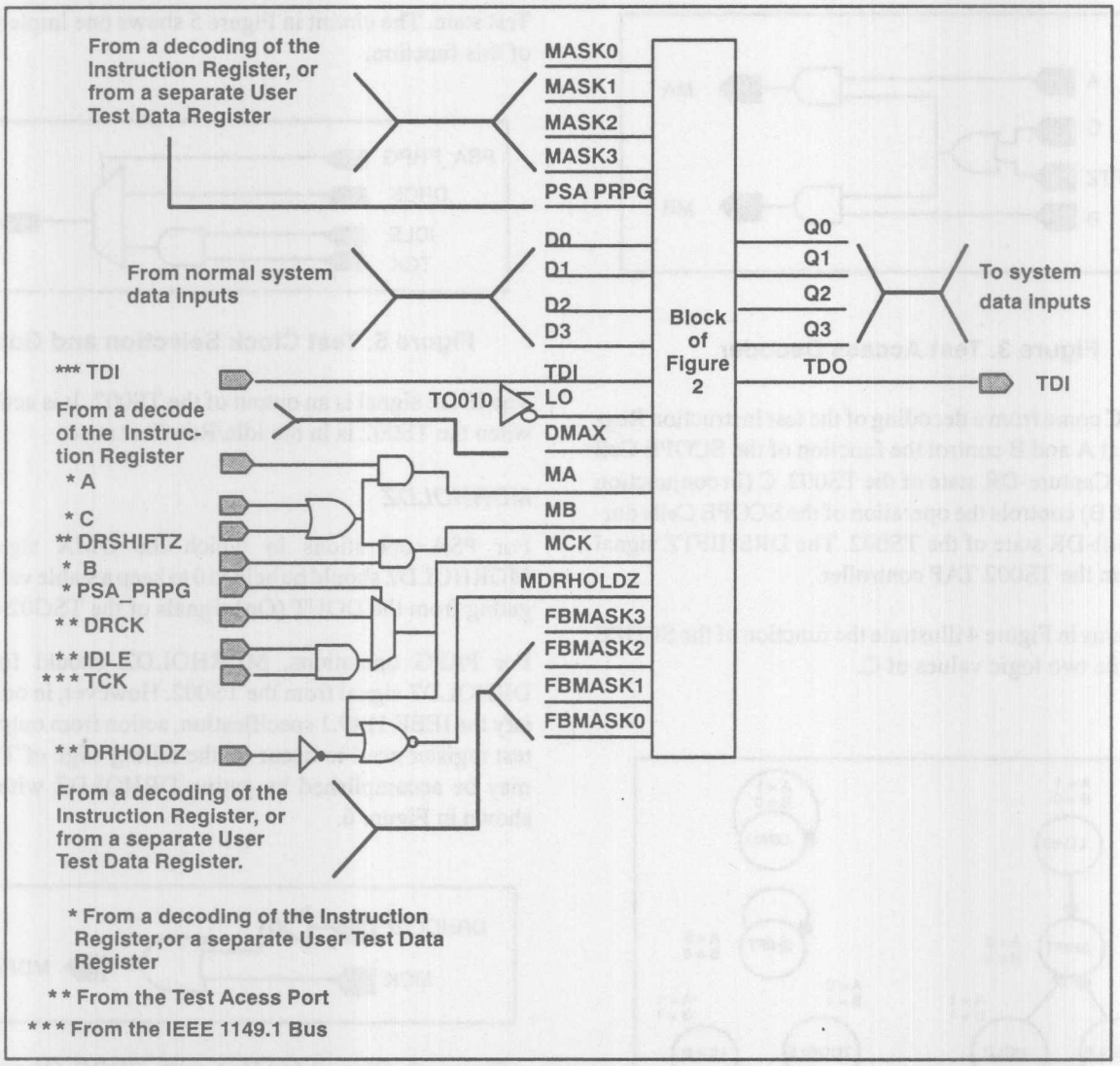


Figure 7. Interconnect Diagram

Description of Output Signals

Qn

The Qn data outputs from the TSG02 cells follow the functional (normal) Dn inputs when the LFSR is configured in its normal transparent mode (when DMX is low or 0). The Qn data outputs can be held to a specific scanned in value or generate pseudo-random patterns if commanded to by the test instruction. The Qn outputs should be connected to the normal logic block outputs or to the logic block inputs (depending on whether this LFSR is on the "input" or "output" side of the logic block).

TDO

TDO should be connected to the TDI input of another scan register, or to a data register multiplexer input on its way to the TDO output of the ASIC.

Practical Application Notes

Trade-Offs

Due to the extra circuitry involved to implement an LFSR, a designer should trade off this test solution against the simpler (smaller) boundary-scan solution.

Internal and External Testability

Once committed to the LFSR structure, only a few more gates are needed to configure it as a PSA test register or a PRPG test register. Keep this in mind when deciding on the system testability functions this ASIC may help you solve once mounted in a system.

Considerations

Consult a book about LFSR structures and their use in testability before choosing feedback taps or the length of the LFSR. Also consider that a seed (starting) value must be scanned into the LFSR before the test operation (PSA or PRPG) is performed. A book on testability will help you decide on an appropriate seed value. Different seed values result in different signatures and output pattern sequences.

Scan Access to Register

Consider the fact that after a PSA (or even PRPG) operation is performed, the resulting signature will need to be scanned out for verification against a known good signature. Figure 3 illustrates the transitions of the TSG00 Cell based on MA and MB input signals to the LFSR structure. The HOLD-then-SHIFT operation would be a good instruction to be able to load into the test Instruction Register in order to scan out the signature held in the LFSR.

Order of Operations

To perform the PSA/PRPG operation, first load the starting seed values into the test register, then scan in the BIST or other suitable instruction into the Instruction Register. Upon entering the Run-Test/Idle state of the Test Access Port, the operations should begin. To observe the test results, scan out the PSA test register contents and compare it with the known good value.

Scan-Based Design Verification – An Alternative Approach

by Pete Fleming and Don McClean

This paper was presented at the ATE and Instrumentation Conference West, 1990.

Introduction

Electronic digital design has been characterized by rapidly increasing performance and functional density at reduced cost. End-users have realized these benefits in the form of smaller, more-reliable, more-capable products with sharp cost drop-offs over time. Continued advances in digital technology will require further miniaturization of geometries that well may yield diminishing returns as we outpace the capabilities of debug and test equipment.

Present-day board designs are incorporating surface-mount technology to increase the equivalent functionality per unit area. Manufacturers are moving toward slim-profile packages and attempting to reduce the space between parts to pack more "power per picoacre." Pin-to-pin spacing on catalog devices is approaching 25 mils, with much smaller spacing on many custom and high-pincount ICs. Tape-automated bonding will reduce geometries further, permitting incredibly dense circuit cards to exist. More and more use of complex submicron Application-Specific Integrated Circuits (ASICs) and custom devices is occurring.

While these trends help produce smaller, better products, they work directly against the objectives of validating the design and proper manufacture of the product itself. As ASICs absorb the functionality of multiple devices, entire sets of bus and control signals disappear into the internals of ICs. The complexity associated with validating the design functionality of an IC or board increases exponentially. The geometries of boards defy the state of the art in fixturing technology, including the use of handheld and clip-on probes.

The result has been major concern over continued use of existing approaches to observe and control digital designs. Anticipated benefits in cost and performance are endangered by escalating test costs and product development cycles.

Test Standardization Efforts

In response to the emerging crisis, test engineers in Europe mobilized in 1985 to form the European Test Action Group (ETAG). Initiated by Phillips, this ad hoc organization began promoting a test technique for manufacturing test called "boundary scan." Boundary scan attempts to overcome the loss of physical access (especially for "bed-of-nails" fixtures) by embedding virtual test points around the periphery of a chip, hence, the name boundary scan.

ETAG began to approach the major silicon suppliers with their proposal, soliciting support in addressing the emerging test problem. As more non-European companies put their weight behind the proposal, ETAG changed their name to the Joint Test Action Group (JTAG). Major involvement from leading semiconductor manufacturers, such as Texas Instruments and Motorola, established JTAG as a credible force in advancing test technology. Simultaneously, an IEEE-sponsored effort to establish standard test buses known as IEEE P1149 emerged in the United States.

When the boundary-scan technical proposal[1] had secured the endorsement of dozens of major electronics firms, JTAG approached the IEEE in an attempt to formalize the ad hoc effort. JTAG was folded under the umbrella of P1149, which subsequently migrated into a series of coupled, but independent, bus definitions known as P1149.n. The furthest defined was the JTAG bus, which was designated IEEE 1149.1. The boundary-scan proposal evolved through several draft revisions before going to ballot in August 1989.[2] The result was an overwhelming endorsement for 1149.1, which achieved formal IEEE approval in early 1990.

Boundary Scan

The goal of boundary scan is to regain lost visibility and control of designs through the inclusion of additional test logic impacting the input/output (I/O) pins of devices. Scannable

flip-flops are multiplexed onto the IC functional data paths, allowing signals to be observed or to be brought to a known state via a four-wire interface.[3] 1149.1 standardizes the four-wire interface to ensure interoperability of devices from multiple vendors (that is, one can scan through a Motorola part to access logic in a TI part). 1149.1-compliant parts can sample or drive their I/O pins to support testing of both the board and the ICs.

In an external test mode, the outputs of a device are controlled to desired states while the inputs of neighboring devices are observed. In this context, neighboring means that two devices are interconnected by a common signal, not that the devices are physically adjacent. The external test mode allows test vectors to be scanned in and out to verify the proper interconnect of ICs on the board. This also allows the testing of nonscannable logic clusters surrounded by scannable devices.

In an internal test mode, the internal silicon is isolated from input pins while test vectors are propagated across it to the output pins. This allows testing of the device logic to the extent that adequate patterns can be devised for application from the I/O points. The 1149.1 architecture supports interfacing to internal scan and other optional test data registers to assist in testing complex ICs.

The boundary-scan architecture also supports a bypass mode that abbreviates the scan path to a single bit when its boundary scan registers or optional data registers are not accessed. An optional device identification register may be included. Other capabilities can be accommodated easily within the boundary scan framework, such as Built-In Self-Test (BIST), pseudorandom pattern generation, signature analysis, and more.

1149.1 devices dedicate four pins to support the industry standard bus. The 1149.1 bus consists of four wires—two to control the scan state of the devices and two to transmit the serial data. These four wires are routed to all scannable devices on the board.

Conventional Debug Techniques

Designers face the challenge of validating the design of their boards (or ICs) even in the face of high complexity. Standard processes such as visual inspection, ohming out interconnect, and verifying power and ground are complicated by the tight geometries, but the real penalty occurs during attempts to confirm functionality. A wide variety of techniques is used to debug designs, but virtually all of them require external instrumentation that connects to or probes the board under test.

Multimeters or logic analyzers are used to determine logic levels for digital signals and are necessarily constrained to observing I/O level signals. Word generators are connected to buses to apply controlled data streams for test. Bus analyzers may be used to monitor states of standard bus protocols, and in-circuit emulators allow control and debug of microprocessor-based designs.

Reliance on these tried-and-true instruments has two disadvantages in the light of advanced packaging techniques.

First, tight geometries and high-speed signals are not amenable to probing and clipping.[4] Engineers and technicians cannot probe fine-pitch designs reliably unless techniques such as stagger pads are used. Frequently, key signals are embedded within the internal logic of a device, inaccessible to contact-dependent instruments. High-speed signals, such as processor signals present in emulator cables, are subject to cross talk and glitching because of the length and layout of the cables. Second, design validation often requires monitoring of the design in unique environments, such as in the end-use environment or in temperature chambers. This usually requires the electronic subsystem to be closed up where access is limited to connectors on the chassis or case. Conventional instruments cannot be used to probe the design.

In these cases, the visibility and control permitted by scannable designs offer alternative techniques for logic validation.

Scan-Based Debug

Emergence of IEEE 1149.1-compatible components will provide an infrastructure of control and visibility with improved capabilities for multiple engineering disciplines. To harness these capabilities, an environment must exist for controlling and manipulating the scan-accessible features. Additionally, designers must not be overwhelmed by the details of the scan itself.[5]

Design validation always has been viewed as a parallel operation whereby the states of multiple signals are changed or viewed as a single event. This parallel- or register-level view of the design is the one with which designers feel comfortable and pervades existing computer-aided engineering (CAE) tools such as logic and fault simulators.

A Practical Experiment

Texas Instruments developed a system-level test bed using the 1149.1 architecture across four printed wiring boards (PWBs) using the scan for debug and integration. Manipulation of the scan was performed using a Scan Control System (SCS). The

particular tool used for this effort is called the Advanced Support System for Emulation and Test (ASSET). ASSET is a PC-based tool that builds a data base representation for tracking and controlling the state of the scan architecture. This allows the burden of dealing with the serial view and current state of the scan paths to be relegated to the computer. ASSET uses configuration files to determine the topology of the scan paths and interfaces to the unit under test via a controller card in the PC expansion slot. ASSET drives the 1149.1 protocol.

An SCS can provide a wide range of capabilities, but the conversion of the serial view to a parallel one is the key feature. The SCS associates functional signals with scannable locations and serves as a "test operating system" to bring functional signals to a known state simultaneously, similar to a conventional parallel process. For example, if one desires to set an address bus to the value >38, certain functional nodes comprising the address bus must assume a specific logic level. A word generator would clip on to the appropriate devices and simultaneously drive the eight address lines to the required value. In contrast, the SCS uses its knowledge of where the same eight signals reside on the scan path to scan them into the desired value and activate them on a common clock edge.

The actual process involves selecting or deselecting the appropriate boundary-scan registers, issuing the command to the affected devices, and scanning in data values in the proper order. The SCS uses a configuration file that acts like a street map, directing the information to the right mailboxes.

The SCS allows signals such as the address bus to be grouped functionally. With the proper user interface, the designer has a PC-based capability to set registers and buses to the value of choice or to sample the value of these elements. Internal signals are equally accessible as long as they reside on the scan path. The power of the computer allows "programming" of the stimulus steps and automated capture of the data for review at the designer's convenience.

Flow

The ideal approach to implementing the debug process was to have the designer develop the debug procedures while the board was being fabricated. The designer was most knowledgeable of the design of the board, including the partitioning of the functions within the design and how the scan architecture was implemented. The designer performed or assisted in the definition of the configuration file that identified the scan topology to the SCS. Next, desired functional capabilities were identified based on the board design.

Example capabilities included:

- Read memory value
- Write memory value
- Initialize VME interface devices
- Reset board
- Write to speech processor

Subroutines or utilities were created with appropriate user interfaces to raise the level of abstraction from serially oriented bit manipulation to register- or bus-level transactions. The engineer would specify the address in memory to write to and data to be written, and the SCS would handle the rest during runtime. Individual subroutines (such as read/write memory) could be nested further to build upload and download routines.

The designers performed typical prechecks of their boards and use the scan to weed out manufacturing errors wherever possible. The functional subroutines then were used to validate the logic design.

Structural Versus Functional Verification

Structural vectors make no attempt to use the hardware in its functional mode, but focus in on toggling signal states to detect classical manufacturing faults (stuck-ats, shorts, opens). The use of the scan greatly simplified this task, eliminating one of the most frustrating aspects of board debug. When a first-pass design fails to perform as expected during verification, two possible causes exist. First, the designer may have made an error in the design of the logic. The primary goal of design verification is to identify and fix these problems. A second possibility exists that the hardware under test has a manufacturing error rather than a design error. Unfortunately, during the initial debug process, neither the design nor the manufacturing database has been validated. Thus, designers often spend hours re-examining their schematics and simulations in search of nonexistent design errors that turn out to be solder splashes or wirewrap miswires.

Before using the functional routines, the interconnect was exercised in an attempt to locate any manufacturing errors. The ability to scan to the appropriate devices first was verified to eliminate any uncertainty in the results. Any problems existing with the configuration file or the actual hardware implementation of the scan were eliminated. The verification of the interconnect was a straightforward task involving only basic knowledge of the board's functional topology. The scan allowed etch or wiring errors to be located easily for signals driven and monitored via the boundary scan.

The functionally-oriented subroutines then were used to exercise the board logic. The sequencing of signals emulated the

normal logic operation to the maximum extent. The tremendous flexibility of these subroutines provided a highly robust debug environment. One clear advantage of the scan was the ability to select or deselect signals for stimulus or monitoring without the need to rewire or move a probe clip physically. This process, which can require 10 minutes using a logic analyzer, was accomplished in less than 1 minute. The number of signals to be controlled or viewed simultaneously was unlimited, as opposed to use of multiple analyzers or multiple passes on a single analyzer moved around the design.

Processing of the results was a mixture of interactive interpretation of the values displayed on the screen with hardcoded compares of scanned-out values.

The Debugger

TI's SCS, ASSET, has a very valuable feature called the debugger mode. The debugger is a highly interactive mode, allowing the engineer valuable control over the hardware in a debug environment. One function of the debugger provides control over the SCS functions themselves, such as changing the scan clock speed or enabling/disabling ASSET's two runtime checks. ASSET provides the ability to verify the integrity of the scan path during each instruction scan to identify any potential corruptions. It also provides a prescan check to prevent any user-identified conditions from occurring. For example, via the chain scan, one could enable conflicting buses simultaneously, potentially damaging the board. The prescan checks would prevent such states from being scanned into the hardware if they had been identified as illegal conditions.

A more powerful function of the debugger is the view function. In view mode, the engineer can open windows for each scannable part. Via these windows, instructions can be issued to individual parts to manipulate the scan-accessible registers. This allows a robust interface to monitor signal states, set register values, turn on pattern generators, develop signatures, or any of the test capabilities one can layer over 1149.1. The windows can group multiple devices into functional elements (such as constructing a view of a 32-bit data bus that physically crosses four 8-bit parts), and can window into multiple boards simultaneously.

Another key function is the embedded logic analyzer/word generator function. This function allows the user to apply stimulus and trace scannable signals in the system in a batch mode and present the results in a waveform display. The designer can select or deselect individual signals to trace and choose a stimulus file for application. The stimulus file could be something manually generated by the designer or potentially, a reused portion of the logic simulation vectors. The stimulus is applied to the actual logic and then displayed on the PC in a waveform view similar to a conventional logic analyzer. Another function allows comparison of the results against an existing data base, such as the logic simulators results or the results of a previous run. Differences between the two runs are highlighted in color on the waveform display.

Limitations

The demonstration system developed by TI incorporated a limited variety of 1149.1-compatible parts, but even this partial implementation yielded encouraging results for improving the debug process. Several limitations were identified for the process and need to be addressed.

First, learning curve has a major impact. Valuable experience was gained in implementing the architecture, but at the expense of time and analysis. Conventional instruments such as logic analyzers were used to augment the process until the scan architecture itself was validated. Because this was the first application of many SCS features, it was necessary to resolve design errors versus problems with the SCS itself. This was similar to the problem mentioned previously of looking for design errors when manufacturing errors exist. However, once the core software for the SCS is fully debugged, the process is greatly simplified.

Coming to grips with programming under the SCS was a new experience for the designers. ASSET's environment is very "C"-like, and the designers came up to speed quickly. Some of the designers had developed code previously for debugging designs, and, thus, felt more comfortable with the software development than others. Using the debugger was very instrument-like, and, once the designers understood how to use it, it was a powerful tool.

The process of verifying the scan path required more effort than anticipated, but this is attributed to the introduction of some new interface parts and the respective software drivers for them within the SCS. In some cases, the designers made errors in the configuration files that caused scannable ICs to be addressed improperly. This is corrected easily with improved documentation on the process. It was necessary, however, to use multimeters and logic analyzers until these problems were identified and overcome.

Another limitation was that of scan, in general. The process of applying and capturing vectors via scan was synchronous to the scan clock and involved the overhead of scanning for each event. Thus, the test steps were slower than conventional instruments and required fully synchronous execution.

Emulation

Another design verification technique applied to microprocessor architectures is emulation. Conventional emulators require the device to be socketed so it can be removed during emulation. A special cable interfaces a hardware emulator (a separate box) into the socket, and software is used to model the execution of the microprocessor. This allows processor boards to be tested, as well as allowing the software hosted on the board to be debugged.

Unfortunately, emulators now must run at speeds too excessive to support the external cabling techniques. Each upgrade to a microprocessor requires a new emulator model to be designed, delaying entry to the market. Customers are not pleased with the additional capital investment required for each emulation system.

Via 1149.1, the capability exists for the majority of emulation features to be implemented in silicon and accessed via the scan bus. An example is TI's family of Digital Signal Processors (DSPs). The TMS320C30 supports many emulation features tied to an earlier bus interface, and future generations of DSPs will use 1149.1 as the port. Ultimately, this will allow board-level emulation, providing tremendous visibility into complex parts and offering powerful capabilities for software debug and test.

Summary

Advanced digital technology is outstripping the capabilities of many conventional testers and instruments because of reliance on physical access (unless painful concessions are given in the packaging area). The formalization of the IEEE 1149.1 Boundary Scan standard will introduce an infrastructure that not only aids and abets the test engineer, but equally assists the design engineer in debug of hardware and software. New scan-based tools can provide the same functionality of existing debug instruments without the burden of probing. This infrastructure is constantly available as long as the access to the scan bus exists, allowing new environments such as temperature chambers to be addressed. Designers will be able to verify the structural correctness of the prototype designs easier, eliminating ambiguity between design and manufacturing errors. Scan-executed sequences provide a viable, robust, interactive method for functionally exercising the design logic and may reduce the effort and cost of the design verification process significantly.

References

[1] *Joint Test Action Group Specification*, Version 2.0.

[2] *Proposed IEEE P1149.1 Standard Test Access Port and Boundary Scan Architecture*, Draft D5, June 20, 1989.

[3] Lee Whetsel. *A Proposed Standard Test Bus and boundary-scan architecture*, Proceedings of the International Conference on Computer Design, October 1989, pp. 330–333.

[4] Andy Halliday, Greg Young, and Al Crouch. *Prototype Testing Simplified by Scannable Buffers and Latches*, Proceedings of the International Test Conference, August 1989, pp. 174–181.

[5] Don McClean and Javier Romeu. *Design for Testability with JTAG Test Methods*, Electronic Design, June 1989.

Standard Test Port and Cells Provide an ASIC Testability Toolkit

by Lee Whetsel

Test standardization activities currently taking place in the electronics industry will have a significant impact on the way companies view and apply test. One of the key technical accomplishments that has recently emerged is a four-wire Test Access Port (TAP) specification developed by the Joint Test Action Group (IEEE 1149.1).

Anticipating the industry acceptance and eventual formalization of the TAP through an appropriate standards organization such as the IEEE, many companies are beginning to implement the TAP in new chip designs.

Within TI, a Corporate Test Standards Committee has been established to create a common test approach and interface between the different groups of TI. One of the areas where test standardization can have a near-term impact is in TI's Application-Specific Integrated Circuits (ASIC) organization.

Standard ASIC test cells being developed will provide ASIC designers with the building blocks for test structures to support chip- and board-level test. This article describes an ASIC testability toolkit consisting of TI standard test cells and scan interface macros, to support ASIC Design-For-Testability (DFT).

Transitioning From Board- to ASIC-Level DFT

In the past, experts in board-level testing have been able to overcome testing problems with creative, ad-hoc board-level test solutions. Most of these ad-hoc test approaches involved adding dedicated test hardware to the board design to partition the circuit into smaller, more easily tested logic blocks. In addition, the printed circuit board would include connectors and test pads to facilitate test access to the circuit by functional and in-circuit testers.

As more board-level designs are being directed into an ASIC environment, test engineers must adapt to a different test approach. Although the partitioning technique applies to either an ASIC or board design, the test access to the partitioned blocks differs significantly. In board designs, it is generally assumed that testers will have access to sections of the design via connectors or test points. However, in an ASIC design, test access must be accomplished using only a minimum number of device pins, and typically in a serial format. Also, in chip design greater emphasis should be placed on Built-In Self-Test (BIST) techniques.

To aid the test engineer in making the transition from board to ASIC designs, standardized test cells and rules for implementation are presently being developed. The following is a description of the standard test cells that will form the basis of a TI ASIC testability toolkit.

TI Test Access Port (TS002)

To support the standard IEEE 1149.1 interface in ASIC designs, a pair of TI test access port (TS002) macros are being developed for slave- and master-interface applications. The slave TS002 will be included in designs that will receive scan control for test access from an external master device. The master TS002 will be included in designs that are required to transmit scan control for test access to one or more slave devices.

In Figure 1, the slave TS002 responds to a IEEE 1149.1 protocol sequence applied via the Test Clock (TCK) and Test Mode Select (TMS) signals, to shift data into either the instruction or data register of the host ASIC via the Test Data Input (TDI) and Test Data Output (TDO) pins. The slave TS002 in combination with the instruction register, provides the test interface required to access the standard ASIC cells residing in the data register of Figure 1.

In Figure 2, the master TS002 issues IEEE 1149.1 scan protocol and serial test data via the TCK, TMS, TDI and TDO sig-

nals, to accomplish testing of external devices incorporating the slave TS002. The reason for designing a master TS002 capability into "smart" ASIC designs is to reduce the amount of board space required for test by eliminating the need for an external IEEE 1149.1 controller IC, interfacing logic, and wiring interconnect.

ASIC Core Scan Test Cells

The following description gives an overview of ASIC test cells directed towards improving core logic testing. The core test cells blend scan test features in with the function logic of commonly used standard cells, such as flip-flops and latches, to provide test access with a minimum impact on silicon overhead.

Scan Test Flip Flops

To provide scan-testable features in edge-sensitive ASIC designs, flip-flops with scan-test capability can be used in place of normal D flip-flops when constructing registers, state machines, counters, etc. The scan-test flip-flop in Figure 3 consists of a D flip-flop and a 2-to-1 multiplexer.

During normal operation, the data register shift enable (DRSHIFTZ) input is set high and the CLK input is controlled by a device clock pin or clock generator. While the DRSHIFTZ input is high, the Normal Data Input (NDI) is coupled to the flip-flop's D input, via the 2-to-1 multiplexer. In this mode, the scan-test flip-flop operates as a normal D flip-flop, and data is transferred from the NDI input to the Normal Data Output (NDO) on the rising edge of the CLK input.

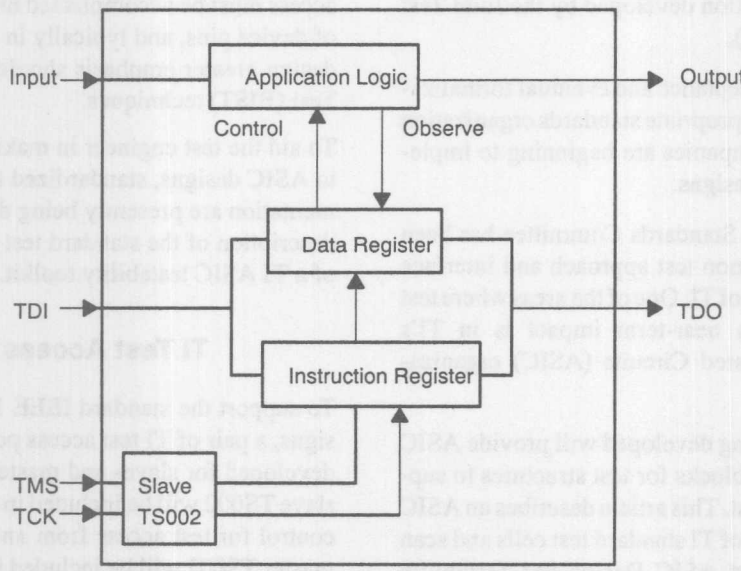


Figure 1. ASIC Slave TS002 Example

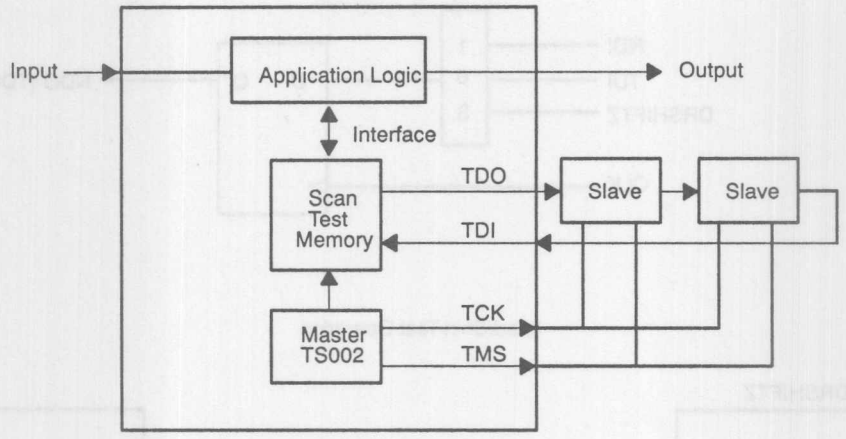


Figure 2. ASIC Master TS002 Example

During test operation, the CLK and DRSHIFTZ inputs of the scan-test flip-flop are controlled by the combination of the TS002 and instruction register of Figure 1 to allow loading and shifting of the scan-test flip-flop. Figure 3 shows that the first CLK of a load/shift test operation causes the D flip-flop to preload the logic level on the NDI input, since DRSHIFTZ is high. Following the first CLK input, the DRSHIFTZ input transitions to a low logic level, causing all subsequent CLKs of the load/shift operation to shift data from the TDI to the NDO (TDO). The load/shift operation allows the logic state of the internal circuit node attached to the NDI to be captured and shifted out for inspection.

In Figure 4, an example application of a scan-test flip-flop is shown. The circuit consists of input combinational logic, register section, and output combinational logic. The circuit receives external control input and issues external control output. An internal feedback path exists between the output and input combinational logic sections.

If normal D flip-flops were used to design this circuit, test controllability would be limited to the patterns applied at the control inputs and test observability would be limited to the response seen at the control outputs. This limited test access may not be sufficient to cause the circuitry to pass through all possible internal register states, and some internal faults may not be detectable.

If scan-test flip-flops are used in this design, as shown in Figure 4, the circuitry can be partitioned during test so that the outputs from the input logic partition are observable, and the inputs to the output logic section are controllable via the scan-test flip-flops. The additional test control and observability from the scan-test flip-flops in combination with the external test control and observability from the control input and output provides the test coverage required to detect all singularly occurring static faults.

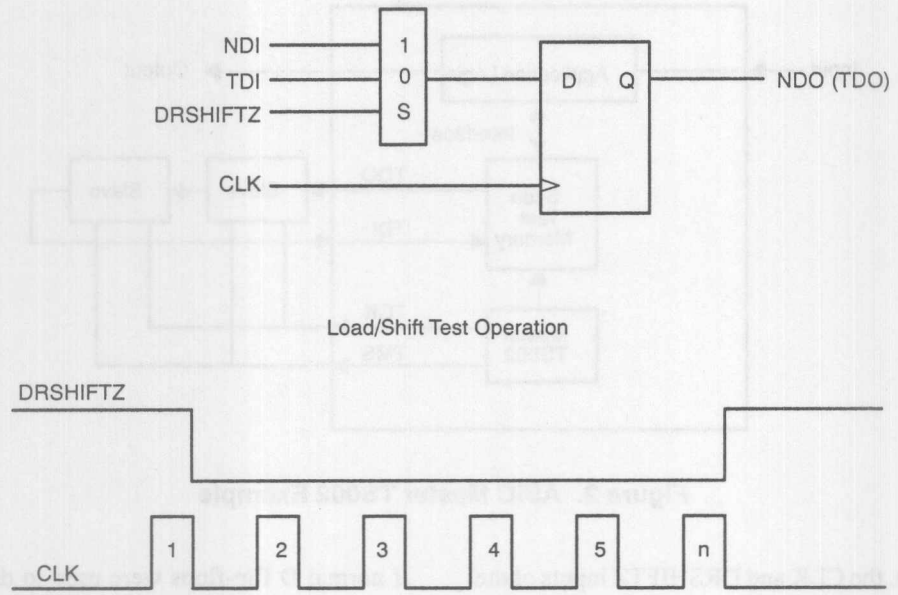


Figure 3. Scan-Test Flip-Flop

Scan Test Latches

To provide scan-testable features in level-sensitive ASIC designs, latches with scan-test capability can be used in place of normal D latches when constructing registers, state machines, counters, etc. The scan-test latch in Figure 5 consists of a dual D-port input latch (L1) and a signal D-port input latch (L2).

During normal operation, the CLK input of L1 and the CK2 input of L2 are controlled by a device clock pin or clock generator. In normal operation, CLK and CK2 operate in the two-phase (a,b), nonoverlapping scheme shown in Figure 5. The CK1 input of L1 is held low during normal operation and has no effect on the functional operation of L1 or L2. When a clock pulse is applied to the CLK input of L1, the logic level on the NDI input is latched at the Q output of L1. Likewise, when a clock pulse is applied to the CK2 input of L2, the logic level on the Q output of L1 is latched at the DCO output of L2. The application of a clock pulse on CLK (phase a) followed by a clock pulse on CK2 (phase b) defines a complete latch update cycle.

During test operation the CLK and CK1 inputs of L1 and the CK2 input of L2 are controlled by the combination of the TS002 and instruction register (Figure 1) to allow loading and shifting of the scan-test latch. In Figure 5, the beginning of a load/shift test operation occurs when the CLK input of L1 is activated, causing L1 to preload the logic level of the NDI input. Following the first CLK pulse, the CK2 input of L2 is activated and L2 updates to the present state of L1.

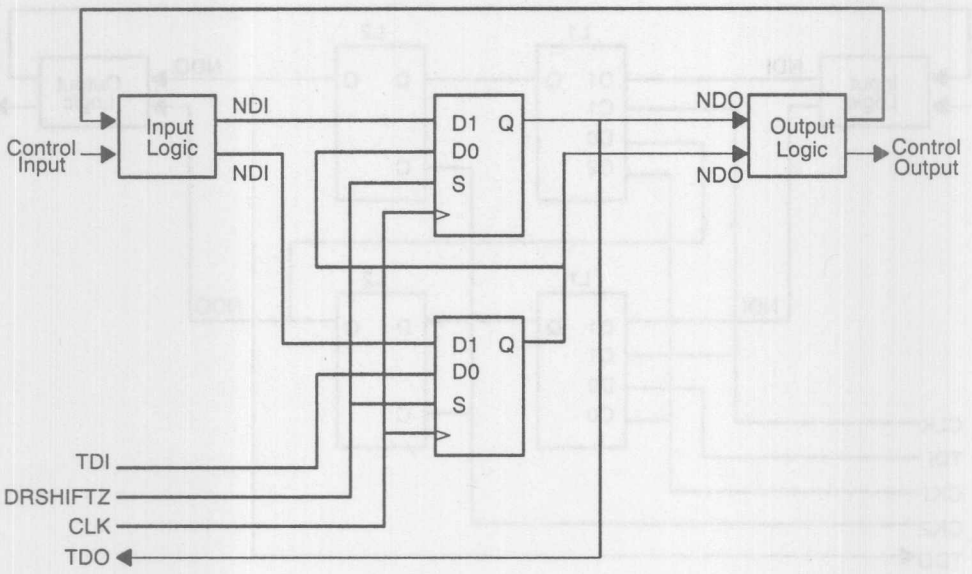


Figure 4. Scan Test Flip-Flop Application

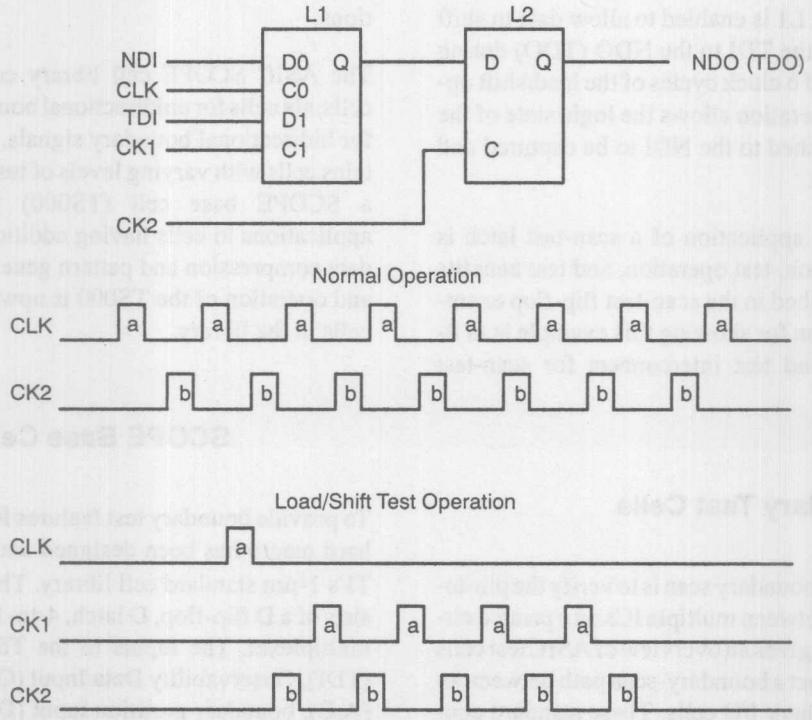


Figure 5. Scan Test Latch

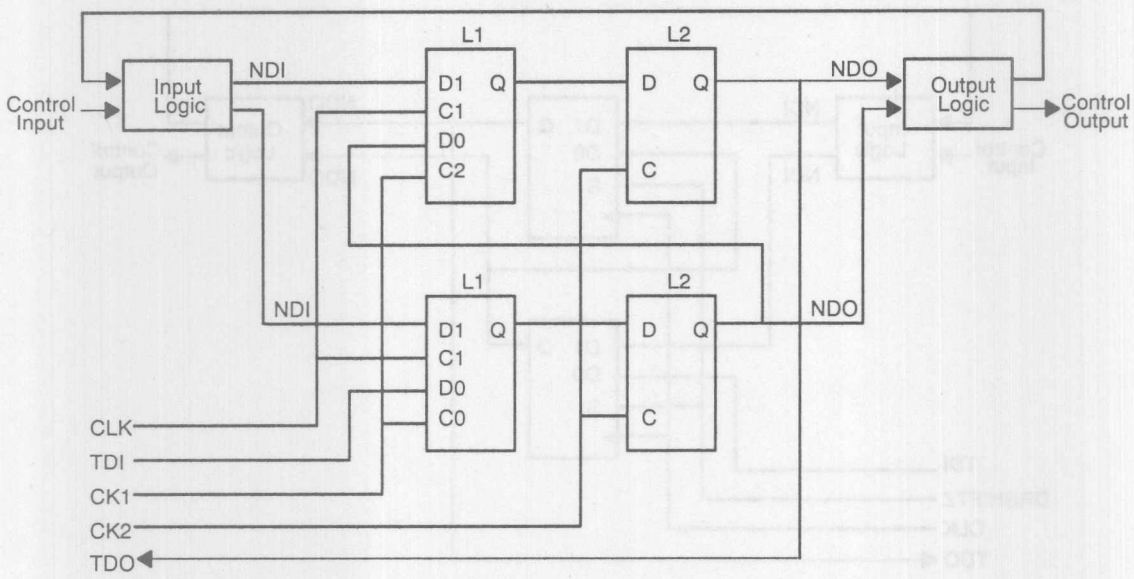


Figure 6. Scan-Test Latch Application

After this first preload operation, the CLK input of L1 remains low and the CK1 input of L1 is enabled to allow data to shift through L1 and L2 from the TDI to the NDO (TDO) during all subsequent phase a and b clock cycles of the load/shift operation. The load/shift operation allows the logic state of the internal circuit node attached to the NDI to be captured and shifted out for inspection.

In Figure 6, an example application of a scan-test latch is shown. The circuit function, test operation, and test benefits are similar to those described in the scan-test flip-flop example application. The reason for showing this example is to illustrate the functional and test interconnect for scan-test latches.

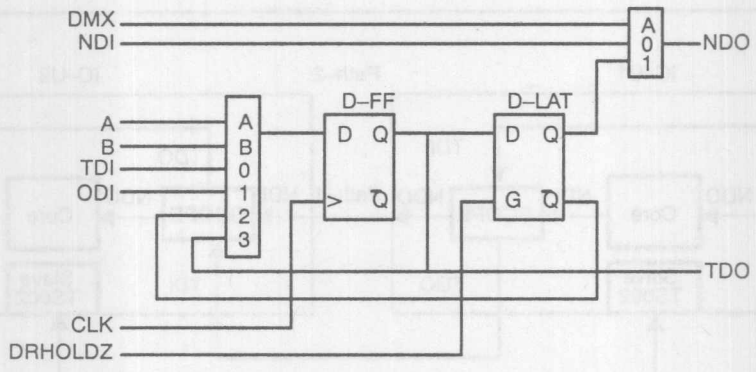
Boundary Test Cells

The basic requirement of boundary scan is to verify the pin-to-pin wiring interconnect between multiple ICs on a printed circuit board. The following gives an overview of ASIC test cells that can be used to construct a boundary-scan path between an IC's core logic and periphery I/O cells. These standard controllability and observability perimeter elements (SCOPE) are dedicated for testing and not blended in with the functionality of an IC, allowing them to be activated during normal operation of the host IC to perform boundary test and shift operations.

The ASIC SCOPE cell library contains 12 boundary-test cells, six cells for unidirectional boundary signals and six cells for bidirectional boundary signals. The SCOPE library contains cells with varying levels of test capability, ranging from a SCOPE base cell (TS000) for basic boundary-test applications to cells having additional test features, such as data compression and pattern generation. The scan interface and operation of the TS000 is upwardly compatible to other cells in the library.

SCOPE Base Cell (TS000)

To provide boundary test features for ASIC designs, a TS000 hard macro has been designed and is currently available in TI's 1-μm standard cell library. The TS000 in Figure 7 consists of a D flip-flop, D latch, 4-to-1 multiplexer, and a 2-to-1 multiplexer. The inputs to the TS000 are Test Data Input (TDI), Observability Data Input (ODI), test command inputs (A,B), boundary partition input (DMX), Normal Data Input (NDI), clock input (CLK), and shadow latch enable input (DRHOLDZ). The outputs from the TS000 are Normal Data Output (NDO), and Test Data Output (TDO).



Note: In boundary-scan applications, ODI and NDI inputs are connected.

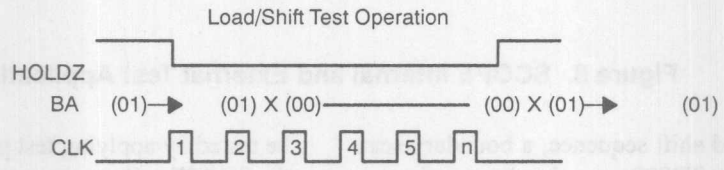


Figure 7. SCOPE Base Cell

The A and B inputs in combination with the 4-to-1 multiplexer allows the TS000 to perform four basic test commands. The test action performed by each test command is described below. The hold command (AB=11) provides a stable state for the TS000 for synchronous design applications.

1. If AB=00, the TS000 shifts data from TDI to TDO during a CLK input.

2. If AB=10, the TS000 loads data from the ODI input during a CLK input (see Note in Figure 7).

3. if AB=01, the TS000 toggles its present state during a CLK input.

4. If AB=11, the TS000 holds its present state during a CLK input.

The TS000's 2-to-1 multiplexer provides the partitioning logic required for boundary testing. In boundary-scan applications, each IC input signal is routed through an input TS000 from the NDI to the NDO, and each IC output signal is routed through an output TS000 from the NDI to the NDO. During normal operation of the host IC, the DMX input to the 2-to-1 multiplexer is set low to provide a direct path between NDI and NDO. During boundary test mode, the DMX input is set high to allow the NDO output to be controlled by the TS000.

When the host IC is in normal operation (DMX=0), a boundary sample operation can be performed by issuing control, via the A and B inputs, to cause the TS000 to load (AB=10) and shift (AB=00) out the logic state of the boundary signal attached to the NDI input (see Note in Figure 7). Also during normal operation, a self-test can be performed on the boundary scan path by issuing control to cause the TS000 to toggle (AB=01) and shift (AB=00) out of its present state. Neither of these test actions affect the normal operation of the host IC.

When the host IC is in a test mode (DMX=1), the boundary signal attached to the NDI input can be observed, and the boundary signal attached to the NDO output can be controlled. During boundary-scan testing, the TS000 will receive a load command to capture the data at the NDI input (see Note in Figure 7). Following the load command, a shift command is issued to:

1. Install the next test-control data to be applied from NDO, and

2. Remove the present test data captured from the NDI input for inspection.

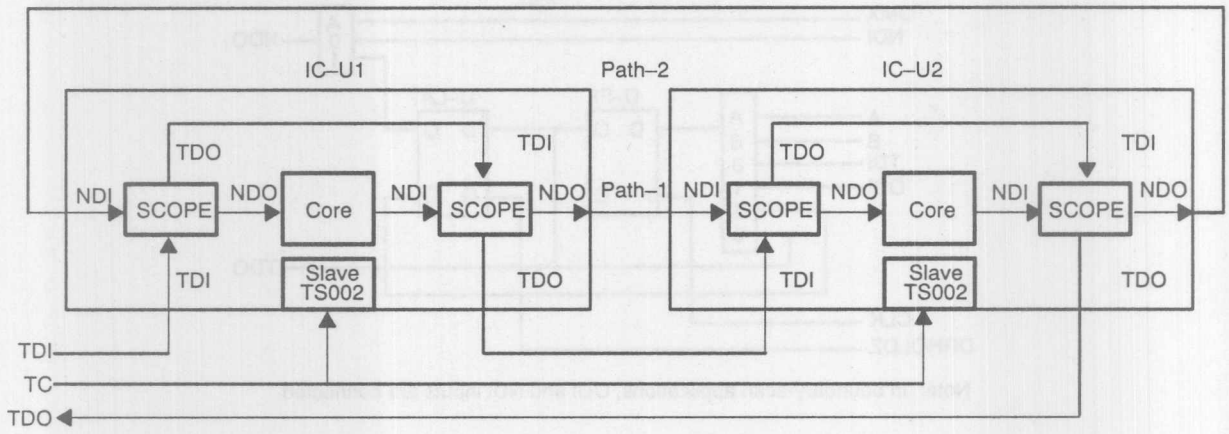


Figure 8. SCOPE Internal and External Test Application

By repeating this load and shift sequence, a boundary scan path consisting of multiple TS000s can simultaneously perform an internal test of the IC's core logic and an external test of the wiring interconnect between neighboring components on a board.

The design of the TS000 includes a shadow latch and a high (active) latch-enable input (DRHOLDZ). The shadow latch and enable input provides two key boundary-scan features. First, the inclusion of a shadow latch within each TS000 provides two separate memory elements for testing purposes — one to observe the TS000 ODI input (D flip-flop) and one to control the TS000 NDO output (shadow latch). Secondly, the shadow latch prevents the NDO output from rippling during shift commands. If the NDO were allowed to ripple during a shift operation, the state of asynchronous logic connected to the NDO output could be altered.

In Figure 8, an example circuit design consisting of two ICs (U1 and U2) is shown. Both U1 and U2 are designed to include a SCOPE cell on each input and output boundary signal. The SCOPE cells are interconnected around the boundary of each IC to form a single scan path from the primary TDI pin to the primary TDO pin. The TDO of U1 is wired to the TDI of U2 to combine the scan paths of both ICs into one circuit-level scan path. Test control (TC) is applied to each IC via the slave TS002 during boundary test and shift operations. During normal operation, the SCOPE cells have no affect on the circuit and both ICs interact together normally via path 1 and path 2.

When the SCOPE cells are placed in test mode, the normal operation of the circuit is inhibited. In the test mode, path 1 can be tested by applying test patterns from the NDO outputs of the SCOPE cells on the output boundary of U1 and observing the NDI inputs of the SCOPE cells on the input boundary of U2. Likewise, path 2 can be tested by applying test patterns from the NDO outputs of the SCOPE cells on the output boundary of U2 and observing the NDI inputs of the SCOPE cells on the input boundary of U1. This test is referred to as external wiring interconnect testing and is the primary reason for boundary scan.

While an external test is being performed on the external wiring paths between U1 and U2, the internal core logic of each IC can also be tested. The core logic of U1 is tested by applying test patterns from the NDO outputs of the SCOPE cells on the input boundary of U1 and observing the NDI inputs of the SCOPE cells on the output boundary of U1.

Likewise, the core logic of U2 is tested by applying test patterns from the NDO outputs of the SCOPE cells on the input boundary of U2 and observing the NDI inputs of the SCOPE cells on the output boundary of U2. If internal core testing is not required, the NDO outputs of the SCOPE cells on the input boundary of U1 and U2 should apply a test pattern that places the core logic in a safe state while external testing is being performed.

Although simultaneous internal and external testing is achievable, in most cases the number of test patterns required to fully test the internal core logic via boundary scan is not acceptable due to minimum test time requirements. It is preferred that full system testing of the core be performed by another means such as Built-In Self-Test (BIST).

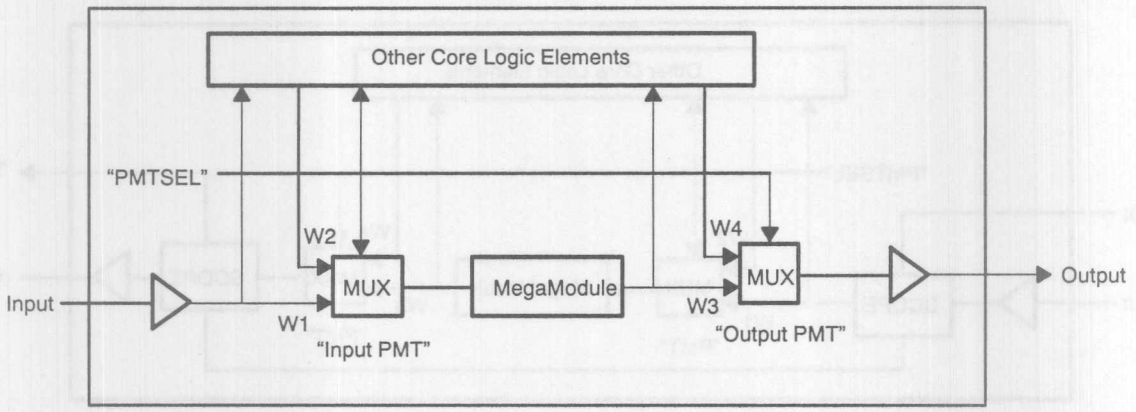


Figure 9. Parallel Module Test (PMT) Cell Application

If the core logic of U1 or U2 consists of memory or combinational logic, SCOPE cells with pattern-generation capabilities can be used at the input boundary, and SCOPE cells with data-compression capabilities can be used at the output boundary to provide a boundary test structure adapted for BIST applications that can quickly verify the core logic using signature analysis.

MegaModule Testing

Standard cell designs are incorporating very large library cells commonly referred to as MegaModules. MegaModules such as microcomputers, UARTs, and large memories present a problem in ASIC testing in that most are not designed with scan test capability and must be tested by applying test patterns at the MegaModule I/O boundary. While testing packaged microcomputers or large memories with automatic test equipment and component fixtures is usually no problem, testing similar devices embedded in an ASIC design is difficult due to restricted test access. Boundary-scan techniques are a potential solution to the MegaModule test access problem; however, the test patterns must be applied serially, which significantly extends the test time of the ASIC product.

To overcome the problems of testing ASIC MegaModules, a Parallel Module Test (PMT) scheme has been developed that allows test patterns to be applied to a MegaModule(s) via the ASIC package pins. PMT cells have been developed that can be inserted in series with a MegaModule's normal input and output signals, to allow reconfiguring the ASIC wiring interconnect of a MegaModule from its normal functional routing path to a parallel module test access path.

In Figure 9 an ASIC design consisting of a MegaModule and other core logic elements is illustrated. A PMT cell is inserted at the input and at the output of the MegaModule to facilitate test. The PMT cell is basically a multiplexer that allows modifying the input connection to, and output connection from, a MegaModule. The path selection of the PMT cell is controlled by a PMT select (PMTSEL) signal from the instruction register of Figure 1.

During test mode, the PMTSEL signal will be set so that signal W1 from the ASIC input pin is coupled directly to the MegaModule input via the input PMT cell, and signal W3 from the MegaModule output is coupled directly to the ASIC output pin via the output PMT cell. In the test mode, the PMTSEL signal will be routed to all elements normally driven by W1 and W3 to shield them from any illegal conditions that may occur during the application of the MegaModule test patterns. In this test configuration, parallel test patterns can be applied directly to the target MegaModule from the package pins.

This MegaModule test approach is very straightforward and the ASIC test time is reduced significantly. Usually, test patterns for MegaModules are readily available from equivalent standard packaged components and can be easily reapplied inside the ASIC via the PMT cells. The only problem that may occur with this approach is when the number of MegaModule I/Os exceed the number of ASIC package pins; however, with the current trend of larger ASIC packaging requirement, this problem should not occur that often.

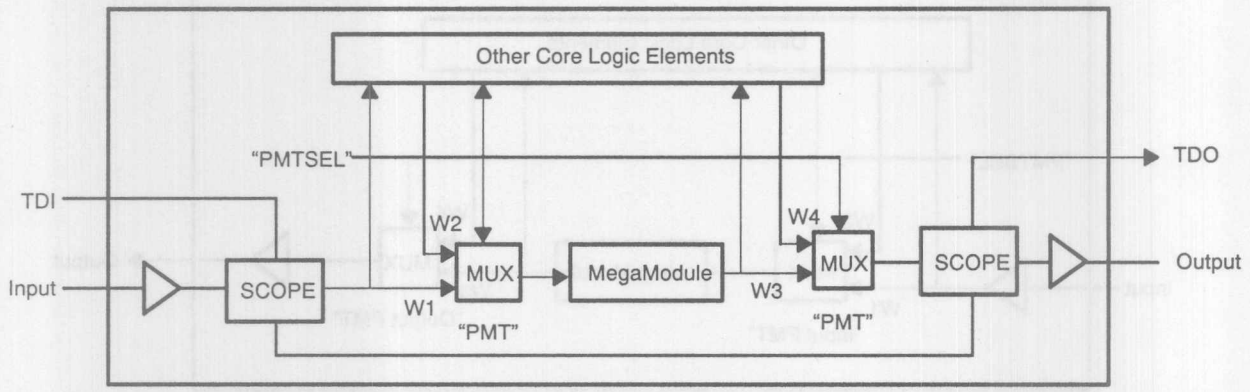


Figure 10. Parallel Module Test Via SCOPE Boundary Test Cells

In some instances the PMT cells and SCOPE cells may be used in combination to allow serial test access to a MegaModule. An example application of an ASIC design using both the PMT and SCOPE cells is shown in Figure 10. This technique allows test patterns to be reapplied to a MegaModule after the IC has been mounted on a board assembly, via a slave TS002.

Conclusion

The ASIC testability toolkit provides the designers with the types of cells and interfaces required to meet their ASIC test requirements. Board test and design engineers will be more at ease in designing ASICs when they become familiar with the benefits and capabilities of a standardized set of test cells.

The next step in improving ASIC testability is to develop standard test architectures, instructions, rules and software support utilities to further enhance the usefulness of a testability toolkit approach.

System Testability Using Standard Logic

by R. J. Morgan

Introduction

The increasing complexity and density of digital systems can be attributed to several factors, the most important of those being the use of more sophisticated ASICs and microprocessors, and a steadily increasing move towards surface-mount packaging. While these advances serve to improve system performance and decrease the physical size of printed-circuit boards, they complicate the task of system test. Access to test nodes using bed-of-nails testing is reduced or eliminated, and the modern automated test equipment necessary to fully exercise complex chips can be prohibitively expensive. One method of circumventing these problems is to use a boundary-scan test method, which can control and observe any node in a system through a dedicated test bus. An IEEE standard currently under development defines a four-wire test bus and protocol that implements a boundary-scan methodology. Standard logic functions, called SCOPE (System Controllability, Observability, and Partitioning Environment) Octals that incorporate this four-wire bus can be strategically placed in a digital system design to greatly enhance overall testability in areas from design and prototype debug to final test and field service of production systems. This paper presents some examples that illustrate the expanded test capabilities available.

IEEE 1149.1

The IEEE 1149.1 document defines a four-wire boundary-scan test protocol that can be implemented in any digital integrated circuit (see Reference 1 for details of the boundary-scan methodology). Chips designed to comply with this protocol can be interconnected to form one or more serial shift register chains, or scan paths, within a system. Texas Instruments has designed four octal integrated circuits that adhere to the 1149.1 protocol, and include many additional capabilities that increase their effectiveness in board, subsystem, or system test.

An Overview of SCOPE/Boundary Scan

Boundary scan is a design-for-testability method that allows signals to be captured and/or forced at the I/O pins of a digital device. All test data is serially shifted from the test data input to the test data output through some register in the device specifically designated for boundary-scan operations. The current instruction defines which of the registers in the device are connected between the test data input and test data output, and only one register at a time may be connected and accessed.

A primary advantage of boundary-scan testing is that the only physical access required is the four-wire test bus. Through this test bus the user can access any testable node or signal in the system. The boundary-scan method has built-in safeguards to make it relatively fault tolerant, and the test protocol allows for a hierarchical approach to test, in which the system is subdivided into stand-alone subsystems in whatever manner is convenient for the design and test engineers.

SCOPE Octals

A block diagram for a SCOPE Octal, the SN74BCT8244, is shown in Figure 1. The device operates in two modes, normal and test. In the normal mode, the '8244 functions identically to an SN74F244, an eight-wide noninverting buffer. In the test mode, the additional circuitry can be activated to perform a specified test operation.

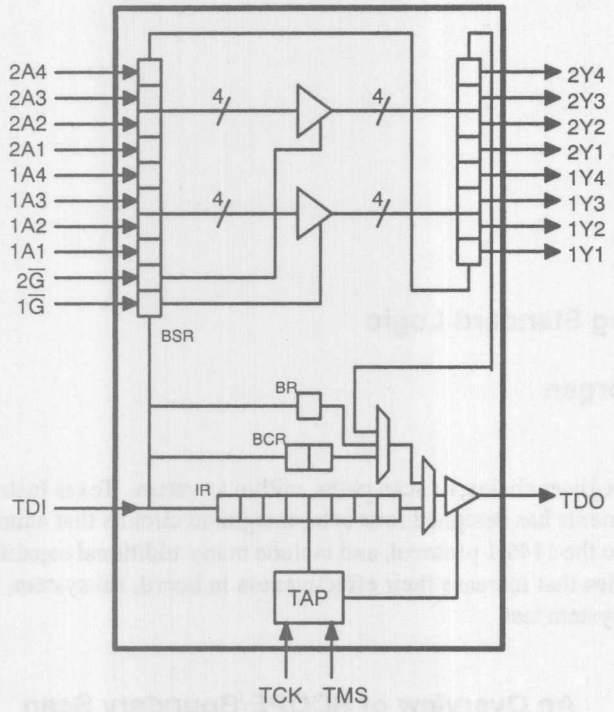


Figure 1. SN74BCT8244 Block Diagram

From Figure 1, it can be seen that the chip contains four serial registers, with each square block representing one bit of the register. One register is the 8-bit Instruction Register (IR), which determines the test operation to be performed. The other three are data registers and are used to capture, load, and shift the serial test information. The 18-bit Boundary-Scan Register (BSR) contains one Boundary-Scan Cell (BSC) for each functional input and output on the device. The two-bit Boundary Control Register (BCR) is an indirect instruction register and is used to implement special test operations not part of the standard SCOPE instruction set. The 1-bit Bypass Register (BR) is used to effectively remove the device from the scan path by providing a short (one clock cycle) delay through the chip. Operation of the test circuitry is synchronous to the Test Clock (TCK). The '8244 advances through its state machine (Figure 2) according to the value of the Test Mode Select (TMS) pin at each TCK rising edge. Inputs are captured on the rising edge of TCK, and outputs change on the falling edge of TCK. Control instructions are issued by the Test Access Port (TAP). The TDI and TDO pins are the serial test data input and output pins, respectively. By connecting the TDO pin of one device to the TDI pin of the next device in the scan path, a serial chain is formed through which all information is passed.

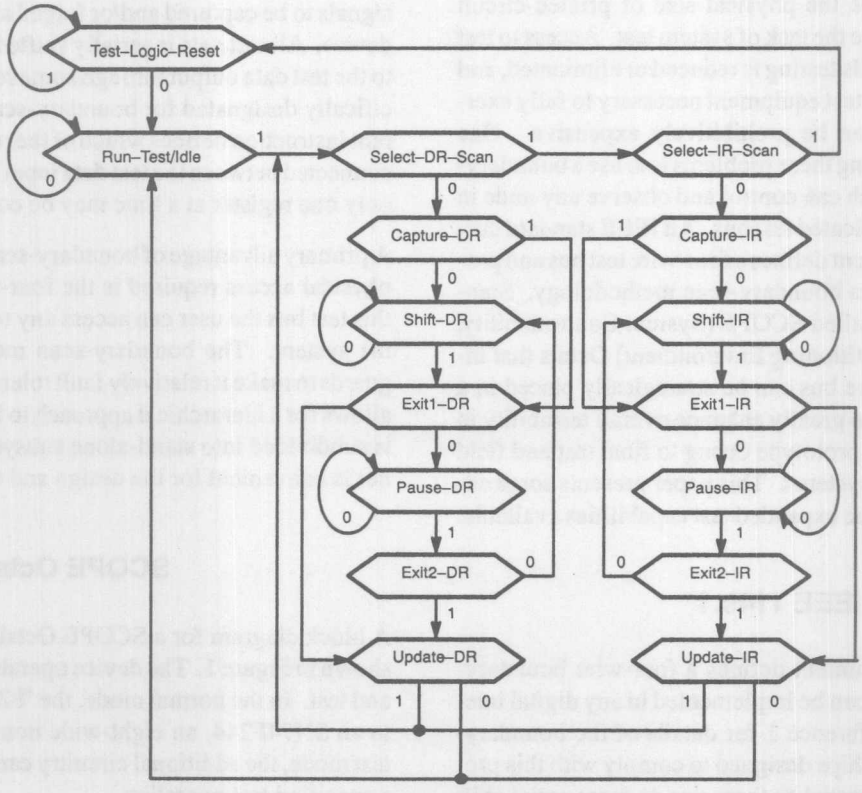


Figure 2. TAP State Diagram

The SCOPE Octals conform to the IEEE 1149.1 protocol and will perform the mandatory instructions of the document. Several other test operations, part of TI's SCOPE instruction set, are also implemented to greatly enhance the testing capabilities of the devices. The instructions implemented in the SCOPE octals, along with a brief description of the operation(s) performed, are as follows:

EXTEST — Data appearing at device inputs is captured. Data previously loaded into the output BSCs is forced.

SAMPLE — Data appearing at the device inputs and outputs is captured without affecting the normal operation of the device.

INTEST — Data previously loaded into the input BSCs is applied to the device's internal logic, and the result is captured at the output BSCs. Data previously loaded into the output BSCs is forced.

BYPASS — The device operates normally and the one-bit bypass register is selected in the scan path.

TRIBYP — Device outputs are placed in the high-impedance state.

SETBYP — The data in the boundary-scan register is applied to the functional inputs and forced from the functional outputs.

RUNT — This instruction tells the device to run the boundary test specified in the boundary control register. These tests include Pseudo-Random Pattern Generation (PRPG), Parallel Signature Analysis (PSA), a combination PRPG/PSA, and output toggling.

READBN — The BSR is placed in the scan path, but no preloading of data takes place prior to shifting.

CELLTST — The contents of the BSC latches are inverted. This is a self-test feature that exercises most of the logic in the BSCs.

TOPHIP — The device outputs are toggled on each TCK falling edge.

SCANCN — The boundary control register is placed in the scan path. This operation is typically performed prior to loading the RUNT instruction.

Due to space limitations, the detailed operations of the SCOPE Octals are not presented here. For more information on the functional and parametric characteristics of the devices, Reference 2 should be consulted.

When in the normal mode, the SCOPE Octals provide high-performance, low-power bus interface functionality. Some of the performance highlights include:

- fabricated in high-speed BiCMOS technology.
- high drive (I_{OH} = –15 mA, I_{OL} = 64 mA).
- low I_{CC} (<10 mA).

In addition to the 'BCT8244, three other devices incorporating the boundary-scan circuitry are available:

- SN74BCT8245 Octal noninverting bus transceiver.
- SN74BCT8373 Octal noninverting D-type latch.
- SN74BCT8374 Octal noninverting D-type flip-flop.

The pinouts for all four SCOPE Octals are shown in Figure 3. Note that the inclusion of the SCOPE circuitry requires the addition of four package pins to the devices, which correspond to the four signals of the test bus.

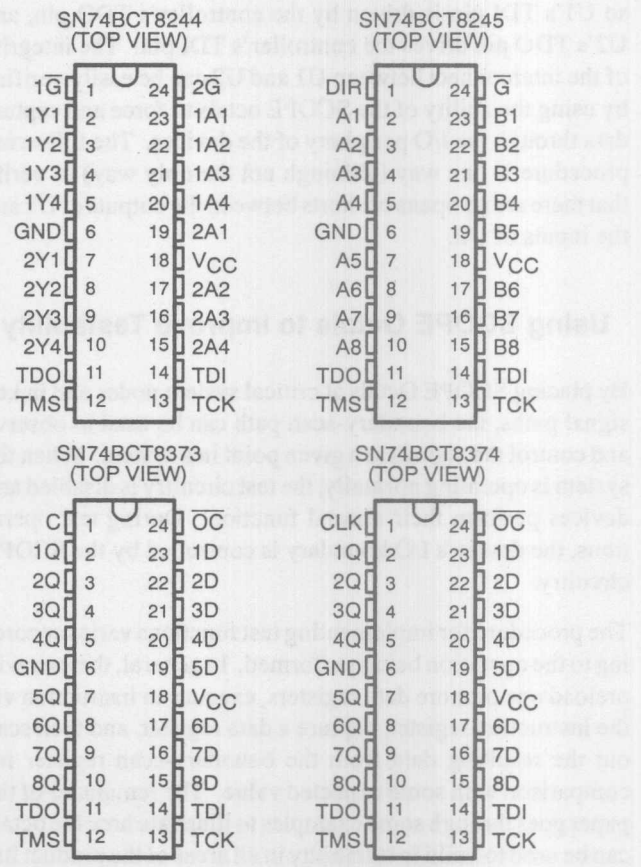


Figure 3. SCOPE Octal Pinouts

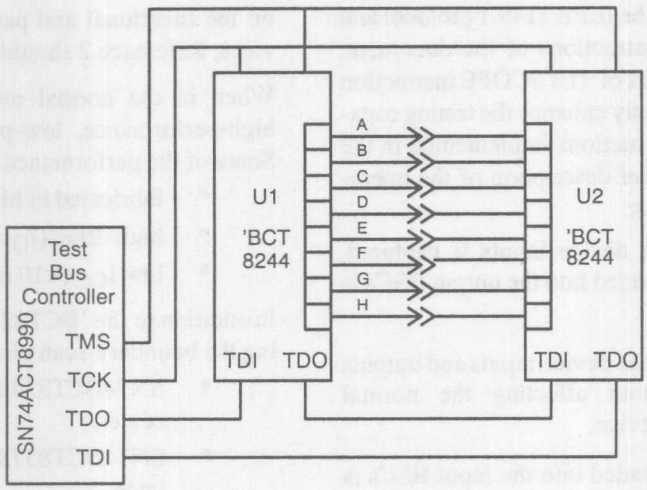


Figure 4. Simple Two-Device Scan Path

In the example of Figure 4, U1 is buffering data being received by U2. The scan path in this case consists of only U1 and U2, so U1's TDI pin is driven by the controller's TDO pin, and U2's TDO pin drives the controller's TDI pin. The integrity of the interconnect between U1 and U2 can be easily verified by using the ability of the SCOPE octals to force and capture data through the I/O periphery of the devices. The following procedure is one way (although not the only way) to verify that there are no opens or shorts between the outputs of U1 and the inputs of U2.

Using SCOPE Octals to Improve Testability

By placing SCOPE Octals at critical system nodes and in key signal paths, the boundary-scan path can be used to observe and control the signals at a given point in a system. When the system is operating normally, the test circuitry is disabled and devices perform their normal function. During test operations, the device's I/O boundary is controlled by the SCOPE circuitry.

The procedure for implementing test functions varies according to the operation being performed. In general, the user will preload one or more data registers, execute an instruction via the instruction register, capture a data register, and then scan out the resulting data from the boundary-scan register for comparison with some expected value. The remainder of the paper goes through some examples to illustrate how the octals can be used to build in testability in all areas of the product life cycle.

Verifying Wiring Interconnects

Perhaps one of the simplest examples of how boundary scan can be used to improve the testability of a system is by verifying the wiring interconnects (detecting "stuck-at" faults) between ICs on a board or between two boards in a system. Figure 4 shows two 'BCT8244s being used to buffer signals between two separate parts of a system. They could be on either side of an edge connector, separated by PCB trace, or in any of a number of other configurations.

1. Initialize the scan path through a reset operation.
2. Scan all zeroes into the output BSCs of U1. This can be done with any of several instructions. ("Scan" means put the TAP in the appropriate shift state and serially load data through the TDI pin.)
3. Scan the EXTEST instruction into both U1 and U2.
4. Capture the BSR of U2.
5. Scan out for inspection the captured contents of U2's input BSCs, while scanning another pattern into the output BSCs of U1.
6. Repeat steps 4 and 5 for each of the patterns necessary to verify the interconnects.

Step 1 can be accomplished by applying 10 V to the TMS pin, by scanning all zeroes into the BSRs, or by a power-down/power-up sequence. During Step 2, load the output BSCs of U1 with the data that will be applied through the functional outputs.

Table 1. Shorts/Opens Verification

Pattern #	Pattern Forced by U1 (A–G)	Pattern Captured by U2 (A–G)
1	00001111	00*1*01111
2	11110000	11110000
3	00110011	00110011
4	11001100	11*1*01100
5	01010101	01*11000*1
6	10101010	10100010

NOTE: Those bits indicating the presence of a defect appear in *italic* type.

The EXTEST instruction is loaded by putting U1 and U2 into the Shift-IR state (see Figure 2) and scanning the SCOPE opcode for EXTEST (00000000) into the instruction register of both chips. During the Update-IR state, U1 will force through its outputs the data loaded in Step 2. Since EXTEST places the BSR between TDI and TDO, going to the Capture-DR state (Step 4) will load the input BSCs of U2 with the data appearing at its functional inputs.

In step 5 we do two things at once, using the Shift-DR state. The scan path is now 36 bits long, 18 bits from U1's BSR and 18 bits from U2's BSR. The data from the input BSCs of U2 is scanned out to be stored and/or examined. Since all zeroes from U1 were forced, the expected value of the data captured at the input BSCs of U2 is also all zeroes. During this same shift, the output BSCs of U1 are loaded with the next pattern. When passing through the Update-DR state after the shifting is complete, the output BSCs of U1 will force the new pattern through its outputs.

As an example, assume that some wiring defects in the circuit of Figure 4 have caused an open between U1 and U2 on signal C, and a short circuit between signals E and F. Table 1 lists six patterns U1 could force to check for any opens between U1 and U2, or shorts between any two pins, and the data that would be captured by U2 (given the defects of this example).

Logic Verification

An application of two features of the SCOPE octals not included in the IEEE 1149.1 document is illustrated in Figure 5. Pseudo-Random Pattern Generation and parallel signature analysis can be used to verify the logic implementation of a design during the debug, prototype, or manufacturing test operations.

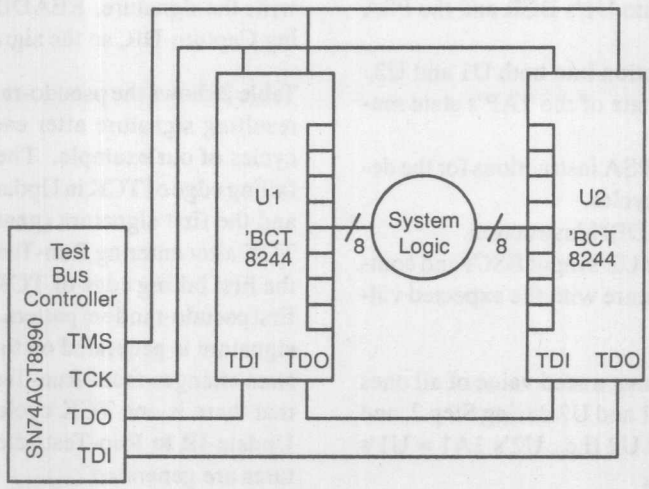


Figure 5. PRPG and PSA

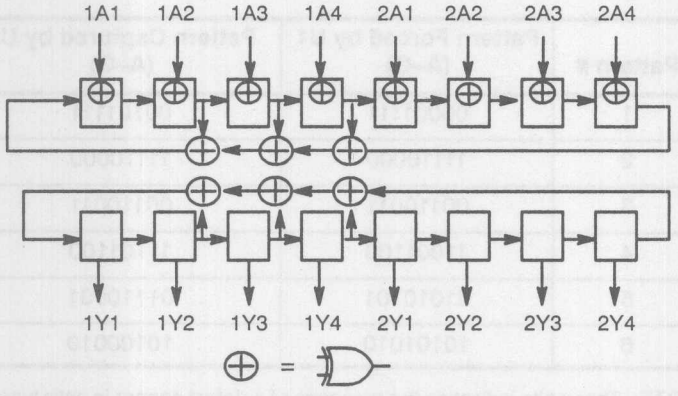


Figure 6. PSA/PRPG Algorithm for the 'BCT8244

During a PRPG or PSA operation, the boundary-scan registers of a device are configured as linear feedback shift registers and will perform either a pattern generation or data compression operation on each TCK cycle. By loading a known seed value (other than all zeroes) into the BSR and knowing the algorithm used, the user can determine the patterns that will be generated and/or the signature resulting from the data compression of inputs.

In order to exercise the system logic shown in Figure 5, the BSCs of U1 can be configured to output pseudo-random patterns and the input BSCs of U2 configured to compress data by performing the following operations:

1. Initialize the scan path.
2. Load the BSRs of both U1 and U2 with the seed values to be used during PRPG and PSA. Any value but all zeroes is acceptable.
3. Scan the SCANCN instruction into both U1 and U2.
4. Scan the PRPG code into U1's BCR and the PSA code into U2's BCR.
5. Scan the RUNT instruction into both U1 and U2.
6. Go the Run-Test/Idle state of the TAP's state machine.
7. Execute the PRPG and PSA instructions for the desired number of TCK cycles.
8. Scan U2 with the READBN instruction.
9. Scan out the contents of U2's input BSCs and compare the resulting signature with the expected value.

To illustrate the concept, assume that a seed value of all ones is loaded into the BSRs of both U1 and U2 during Step 2, and there is no logic between U1 and U2 (i.e., U2's 1A1 = U1's 1Y1, U2's 1A2 = U1's 1Y2, etc.).

During Steps 3–5 the octals are loaded with the proper data and instruction to perform the pattern generation and data-compression operations. SCANCN places the boundary control-register between TDI and TDO so the simultaneous PSA/PRPG code (11) can be loaded using the Shift-DR TAP state. RUNT is loaded into the instruction register, and tells the octals to examine their BCRs and run the test specified. In this example, the simultaneous PSA/PRPG function is being used. Figure 6 shows the algorithm used during the simultaneous PSA/PRPG operation.

After generating sufficient patterns to test the logic, the signature in U2's input BSCs must be examined. This is accomplished using the READBN instruction. It is important that this instruction rather than EXTEST, INTEST, or SAMPLE be used, because while all of these instructions will place the BSR between TDI and TDO for the ensuing Shift-DR TAP state, the other instructions will preload the input BSCs with the current input data during the Capture-DR state and overwrite the signature. READBN does not preload the BSR during Capture-DR, so the signature is preserved.

Table 2 shows the pseudo-random patterns generated, and the resulting signature after each pattern, for the first 15 TCK cycles of our example. The first pattern, applied during the falling edge of TCK in Update-IR, is the seed value of all ones, and the first signature (generated on the first rising edge of TCK after entering Run-Test/Idle) is based on that value. On the first falling edge of TCK after entering Run-Test/Idle the first pseudo-random pattern is generated and applied. The last signature is generated on the rising edge of TCK as the TAP state changes from Run-Test/Idle to Select-DR-Scan. Note that there is one TCK cycle (as the TAP state changes from Update-IR to Run-Test/Idle) in which no patterns or signatures are generated.

Table 2. PRPG/PSA Sequence

Cycle	Pattern After TCK↓ (1Y1–1Y4, 2Y1–2Y4)	Signature After TCK↑ (1A1–1A4, 2A1–2A4)
0 (seed)	11111111	10000000
1	01111111	00111111
2	00111111	10100000
3	10011111	01001111
4	01001111	01101000
5	00100111	00010011
6	00010011	00011010
7	00001001	10000100
8	10000100	11000110
9	01000010	10100001
10	10100001	11110001
11	01010000	00101000
12	00101000	10111100
13	10010100	11001010
14	11001010	00101111
15	11100101	11110010

Although our example (Figure 4) contains no logic between U1 and U2, the same principles are applicable in more complex cases. This method can also be used to verify address decoding to memories, or to apply patterns to the input of a complex ASIC. By placing the octals in the critical paths, the user can control the signals being applied to any node in the system.

System Partitioning

Using SCOPE Octals to buffer key signals can allow for effectively partitioning a system or board during test to remove unneeded or unwanted components. Partitioning a system into separate stand-alone test cells can reduce the number of patterns required to test the section(s) of interest.

In Figure 7, the octals are used to partition a shared-memory configuration in which a digital signal processor and graphics signal processor use the same memory. The 'BCT8245s (U1 and U3) are used to buffer data transmission between the processors and memory, and the 'BCT8373s (U2 and U4) are used as address latches. The four octals are connected in a serial scan path with common TCK and TMS signals.

Using the octals in this configuration creates many testing possibilities that go far beyond simple interconnect verification. For example, one or both processors can be effectively removed from system operation by scanning and executing the TRIBYP instruction on the units buffering it, which will cause the functional outputs of the octals to go into the high-impedance state. This instruction also protects the octals if some type of fixtured testing, such as bed-of-nails test, is to be used by ensuring that the functional outputs are not back-driven.

The status of the octal inputs and outputs can also be captured at any time by using the SAMPLE instruction. This operation does not disturb the normal operation of the devices and will not affect the system, but will capture the logic levels at all inputs and outputs. This information can be scanned out and compared against an expected value. The SAMPLE instruction is also useful for preloading the BSR prior to another test

operation, since the octals will continue to function in a normal mode during the preload scanning.

The circuitry in Figure 7 would also allow the contents of any or all memory locations to be written to or read from. A memory location can be verified by using the EXTEST instruction to force an address using U2 or U4 and capture the data appearing using U1 or U3. This illustrates the ability of the devices to both control and observe the signals to which they are connected. If the entire contents of the memory are known, the PRPG and PSA operations could unload the entire memory using a minimum number of clock cycles, with the final signature being scanned out for inspection.

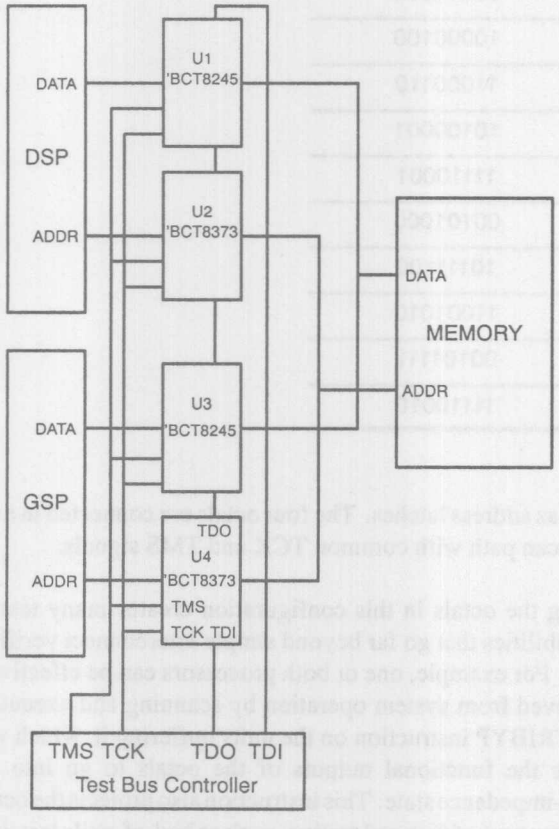


Figure 7. Partitioning a Shared-Memory Configuration

Support Tools

Hardware and software tools are available to reduce the amount of engineering effort necessary to implement boundary scan, including ASSET (Advanced System Support for Emulation and Test), which, when supplied with a configuration file for each device in the scan path, will generate the necessary control and data signals to perform many different tests. Since it is PC-based, ASSET is much less expensive to implement than traditional automated test equipment.

Behavioral models for the SCOPE Octals are also under development that will enable a system designer to evaluate the benefits of the devices before committing a design to hardware prototyping.

Summary

Standard logic functions with boundary-scan circuitry provide a means for thorough testing of digital systems. By using a four-wire test bus compatible with the IEEE 1149.1 protocol, SCOPE Octals allow testability to be built in to a system with complete controllability and observability of any node that can be addressed by a device in the scan path. Boundary-scan techniques are useful in system design, debug, and manufacturing test operations.

Four SCOPE Octals are currently available to provide high-performance bus interface functionality when in the normal mode, and testability enhancement when in the test mode.

References

[1] *Proposed IEEE P1149.1, Standard Test Access Port and Boundary-Scan Architecture*, Draft D6, November 22, 1989.

[2] *SCOPE Octal Data Sheets.*

[3] Lee Whetsel and Greg Young. *Hardware Based Extensions to the JTAG Architecture.*

Using the ASSET Constraint Checker

by Jeffrey Aubert

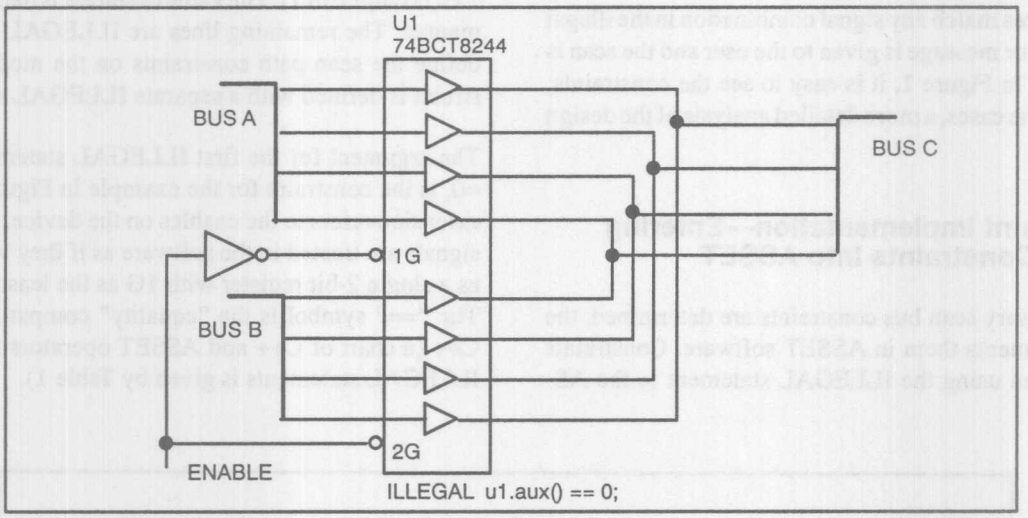


Figure 1. Constraint Example

Introduction

A major concern of digital designers is the prevention of device contention. This is especially true in the debugging and test environment where contention may be present in the design due to test circuitry or software. Contention can be prevented in hardware design by using good engineering practices; however, in a scan bus environment, scan circuitry can override hardware constraints. A mechanism must exist that incorporates constraints against careless or inadvertent (in the case of system debugging) boundary-scan control. This application note shows how to determine and set up boundary-scan constraints in the ASSET software environment. The reader may recognize the similarity between boundary-scan and hardware-constraint determination.

What Is a Constraint?

In boundary-scan test terminology, a constraint is a mechanism that prevents the occurrence of an illegal state. An illegal state is an undesired condition, such as board contention. Illegal states include conditions that would not occur under normal circuit operation. These type of conditions are circuit dependent. Since contention problems are common to most designs, preventing contention will be the focus of most of this paper. Contention occurs when two or more device outputs are asserted on a common signal. Figure 1 shows a simple example of a TI SCOPE octal component ('BCT8244), used to multiplex the values of two 4-bit buses, A and B, to a third bus, C.

In normal component operation, i.e., no scan bus assertion on the outputs, contention between the two input buses is prevented by the inversion of the bus enable signal. When the en-

able signal is a logic one, bus A is connected to bus C; otherwise, bus B is connected to bus C. The hardware constraint is the inverter in the enable logic. When the octal is placed in test mode, external control can be overridden, allowing for the possible assertion of both A and B onto bus C by the scan bus. The constraint for scan is the prohibition of simultaneously asserting both enable signals.

Scan bus constraint is provided in software with the ASSET software package. ASSET provides a constraint checker that compares the data values being scanned with a set of "illegal" scan signal values. These illegal signal values would cause undesired conditions on the board, such as contention, if they were sent to the devices on the scan bus. If the values being sent to the devices match any signal combination in the illegal value set, an error message is given to the user and the scan is not performed. In Figure 1, it is easy to see the constraints; however, in some cases, a more detailed analysis of the design is required.

Constraint Implementation—Entering Constraints Into ASSET

Once the necessary scan bus constraints are determined, the designer implements them in ASSET software. Constraints are implemented using the ILLEGAL statement in the ASSET module configuration file. The module configuration file defines the structure of the scan path for a board (or for an ASIC). An example of a simple configuration file is illustrated by Figure 2.

The file begins with the declaration of my_brd as a module. The next three lines assign device types to the reference designations for scannable components. The order of the devices in the scan path is defined by the MODPATH statement. The SIGNAL statements assign names to the input and output pins of a scan component. The constant membanks is defined as the u7.output(7,0). This is the output of the u7 (a 'BCT8244) with output 7 as the most significant bit (1Y1 is output bit 0 and 2Y4 is output bit 7). The value of selects is defined in a similar manner. The remaining lines are ILLEGAL statements that define the scan path constraints on the module. Each constraint is defined with a separate ILLEGAL statement.

The argument for the first ILLEGAL statement, U1.aux()= =0, is the constraint for the example in Figure 1. The "aux" extension refers to the enables on the device. The two enable signals are treated in the software as if they were configured as a single 2-bit register with 1G as the least-significant bit. The "==" symbol is the "equality" comparison operator in C++ (a chart of C++ and ASSET operators that are used in ILLEGAL statements is given by Table 1).

```
CLASS my_brd:module;          // Declare my_brd as a module (board)
T8244 u1, u2, u7;             // Indicate the type of scan components
T8245 u3, u4, u5, u6;         // used and give their reference
T8374 u8;                     // designators
MODPATH brdpath u1, u2, u3, u4, u5, u6, u7; // Define the scan path order
SIGNAL membanks u7.output(7,0); // Assign names to device
SIGNAL selects u2.output(4,3);  // outputs
ILLEGAL u1.aux() == 0;        // Do not enable both halves of U1
ILLEGAL !u2.output(4) && !u2.output(3);
ILLEGAL u3.dir() == 1 && u3.g__() == 0 && u5.dir() == 0 && u5.g__() == 0;
ILLEGAL u4.dir() == 1 && u4.g__() == 0 && u6.dir() == 0 && u6.g__() == 0;
```

Figure 2. Example Configuration File With Illegal Statements

Table 1. C++ and ASSET Operators

Operator	Description	Example
==	equal to	`u1.g__() == 0`
!=	not equal	`u1.output(3) != 0`
!=	not equal	`u1.output(3) != 0`
>	greater than	`ios(0) + ios(1) > 1`
<	less than	`ios(0) + ios(1) < 2`
<	less than	`ios(0) + ios(1) < 2`
>=	greater than or equal to	`u2.input[] >= 2`
<=	less than or equal to	`u2.output() <= 2`
&&	Logical AND	`(ios(1) == 1) && (ios(2) == 1)`
||	Logical OR	`(u1.g__() == 0) || (u2.g__() == 0)`
+	Addition	`u1.output(1) + u1.output(2)`
()	Use the data in the ASSET transfer buffer	`u3.output(2) == 2`
[]	Use the data in the ASSET receive buffer	`u4.output[4] != 3`

For both enables to be asserted, U1.aux() must be equal to zero since both signals are active low. If both signals are asserted, the condition in the ILLEGAL statement will be true, causing ASSET to produce an error message and abort the scan transaction.

The symbols used in ILLEGAL statements are shown in Table 1. The relational operators are ==, != ,>, >=, <, and <=, which represent equal to, not equal to, greater than, greater than or equal to, less than, and less than or equal to , respectively. && and || represent logical AND and logical OR. The remaining operators, () and [], cause the most confusion. ASSET maintains two sets of data on the scan path: the data values last received from the octals and the data to be sent to the octals. If it is desired to perform a comparison with the data to be sent to the octals, use the () operator. In the first ILLEGAL statement in Table 1, aux() referred to the enable values that were to be sent to the component. If the operator aux [] was used, then the enable values received from the last scan of the component would be evaluated.

Each ILLEGAL statement is independent of the other ILLEGAL statements, that is, the statements are mutually exclusive. If a set of scan values matches more than one ILLEGAL statement, then only the first ILLEGAL statement with a true condition will be evaluated. At translation, ASSET generates code to prevent the illegal states from being scanned. This relieves the designer from writing constraint implementation software.

C++ code is produced by translating the configuration file in ASSET. The XXX.cpp and XXX.hpp files (where XXX is the module name defined in the class declaration of the configuration file) are generated. The file with the .hpp extension is a C++ header file. The .cpp file is the C++ code required to execute the ASSET debugger. In the example given by Figure 2, the files generated are my_brd.cpp and my_brd.hpp. The generated files are compiled and linked with ASSET libraries and user-code-generated object files to produce an executable file. If no software is added by the user, this file will only execute the ASSET debugger; otherwise, the file will execute the debugger and the user-generated code.

The illegal state information is used by ASSET to generate the illegal function in the .cpp file. Figure 3 shows an example of this routine that was generated from the configuration file in Figure 2. This routine is called by ASSET before each scan to verify that the scanned data will not generate undesirable conditions on the target hardware. If any condition given by an ILLEGAL statement is generated, this routine prints the appropriate error message and does not perform the scan. This is helpful during software development and hardware integration and during the use of the ASSET debugger. The conditions in each IF statement correspond to an ILLEGAL statement in the configuration file. Error messages should be changed by the designer as necessary to more accurately describe the specific illegal condition. It is preferred that the designer determines the majority of board constraints before the initial ASSET compilation in order to minimize code changes and to prevent hardware damage; however, constraints can be modified within the configuration file later in the debugging process if the scan code is partitioned properly. Constraints are modified by changing the corresponding ILLEGAL statement in the configuration file. After the file is modified, it is retranslated, recompiled, and relinked.

Software partitioning is accomplished by writing the user-generated code in a separate file (i.e., other than the files gen-

erated by ASSET). The user code should create a C++ class that is derived from the class generated by ASSET. An example is given in Figure 4. The class my_brdfunc is derived from the class my_brd (for a complete discussion on class derivation, also referred to in C++ as class inheritance, refer to the *Zortech C++ Compiler Reference Manual*). The base class, my_brd, is the class specified in the board configuration file (see the class declaration in Figure 2). The code involving the base class is generated by ASSET from the configuration file. The user code is written to utilize the class my_brdfunc, and it is stored in another file in order to partition the code. This permits the designer to modify the configuration file with the constraint changes without having to alter the user code. As a design is debugged, it is not unusual to discover new constraints or to realize that some of the original constraints were too restrictive. Setting up the code to permit configuration-file modifications relieves the designer from altering ASSET-generated code when changing constraints.

Turning Constraints On and Off

During board debugging and system integration, constraint checking is necessary to prevent contention from scan operations (ASSET code). When using the ASSET debugger, the constraint checker prevents the user from interactively imposing an illegal state. However, when both the board (or system) and accompanying ASSET code have been debugged, constraint checking should not be necessary. Disabling constraint checking speeds up program execution since the scan values are not checked before a scan. It should be noted that constraint checking should only be disabled after the ASSET code has been fully debugged. The constraint checker should always be enabled when the ASSET debugger is executed. Running ASSET with user code that has not been debugged can possibly generate contention between components. In many cases, contention can cause device damage. Besides contention, undesired states could be generated, (such as unexpected states for a state machine) that could cause the system to operate in an strange manner.

In the debugger, constraint checking is turned on and off using the ILLEGAL selection in the SCAN menu window. In software, constraint checking is enabled and disabled using the sc_illegal_chk(x) function in ASSET. The function argument, x, is set to TRUE (logic 1), and the constants TRUE and FALSE are defined in ASSET if scan verification is to be enabled. Setting the argument to FALSE disables constraint checking. An example using this function is shown in Figure 5. In this example, the user wishes to execute a test that has been debugged and is known not to produce any circuit contentions. Since constraint checking is not required, it can be disabled to provide increased code execution speed. This would be important in memory testing, for example. After the test is executed, the test result is checked for any errors. Constraint checking is then re-enabled. ASSET enables error checking by default, so sc_illegal_chk() does not have to be called unless it is desired to disable error checking.

```
my_brd::illegal (void)    // The function illegal belongs to the
{                         // class my_brd
   if(u1.aux() == 0) {                  // Code for ist ILLEGAL
     illegalerror ("u1.aux() == .0"); // statement
     return TRUE;
    }
   if(!u2.output(4) && !u2.output(3)) {  // Code for the 2nd ILLEGAL
     illegalerror ("!u2.output(4) && !u2.output(3)"); // statement
     return TRUE;
    }
   if(u3.dir() == 1 && u3.g__() == 0 && u5.dir() == 0 && u6.g__() == 0) {
     illegalerror ("u3.dir() == 1 && u3.g__() ...");
     return TRUE;
    }
   if(u4.dir() == 1 && u4.g__() == 0 && u6.dir() == 0 && u6.g__() == 0) {
     illegalerror ("u4.dir() == 1 && u4.g__() ...");
     return TRUE;
    }
   return (module::illegal());
}
```

Figure 3. Illegal Function Generated From Example Configuration

```
class my_brdfunc:public  my_brd;
  {
    // my_brd is the base class defined by the board
    // configuration file. Since my_brdfunc is the derived
    // from the base class my_brd, it will have the
    // same characteristics as my_brd.

     public:
        my_brdfunc(char *id);
        func1();              // User-defined functions
         .
         .
         .
        funcN();
   };
```

Figure 4. Example of a User Code Class

```
sc_illegal_chk(FALSE);    // Turn off illegal error checking
                          // to decrease test execution time
err = perform_test_of_interest():
if(err)
  {
     // Error code handling

  }
sc_illegal_chk(TRUE);      // Enable illegal error checking
```

Figure 5. Example of Turning Constraint Checking On and Off

Determining Board Constraints

Scan path board constraints should be determined as the design progresses and all pertinent constraints should be determined before attempting to execute ASSET code. This will minimize the possibility of a contention from scan bus operation. Determining scan bus constraints is similar to determining hardware constraints. In Figure 1, the scan constraints are the same as the constraints implemented in designing the hardware, and the required ILLEGAL statement is straightforward. In many cases, however, the scan constraints are more involved.

Common Circuit Situations that Require Scan Constraints

There are several circuit configurations that appear frequently in digital circuit design. Knowing the constraints for these circuit configurations will provide a basis for determining the constraints for any scan design.

Figure 6 shows a situation where two 'BCT8245 data transceivers are used for buffering a data bus. U1 is used to control the data from a backplane while U2 is used to reduce the number of devices on the local bus and provide a greater degree of board partitioning. U2 could be partitioning memory from I/O devices on the local bus between U1 and U2. The obvious constraint prevents U1 and U2 from driving the local bus at the same time. This will require verifying that the enables of both devices are not asserted when the direction of U1 and U2 is from B to A. The constraint is specified by the ILLEGAL statement in Figure 6. Another illegal state would involve U1 driving the backplane when the backplane is being driven. This constraint would involve monitoring the write signal coming from the backplane (when the board is a bus slave) or the write signal being driven to the backplane when the board is a bus master. If this is a bus master in a multiple master system, the bus arbitration signals should also be used. The monitoring of these control signals should be performed using a 'BCT8244 octal buffer. Since these type of signals are usually buffered from and to the backplane, the octal device will replace the standard TTL buffer that is typically used. Figure 7 illustrates a situation that will often occur on data buses. Two registers, implemented with either 'BCT8373 or 'BCT8374 devices, both drive a common bus. The hardware is designed to permit only one register to drive the bus at a time. The scan bus constraint mechanism must ensure that the two registers will not concurrently drive the bus. The ILLEGAL statement in Figure 7 prevents the assertion of both device enables, preventing simultaneous enabling of the registers. Additional constraints should be provided by including the output of any SCOPE octal devices that buffer corresponding signals from the decode logic.

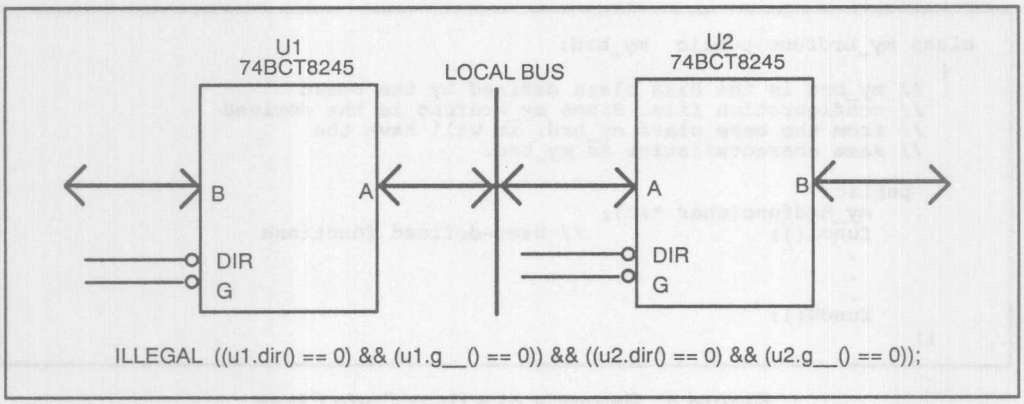


Figure 6. Data Transceivers on a Common Bus

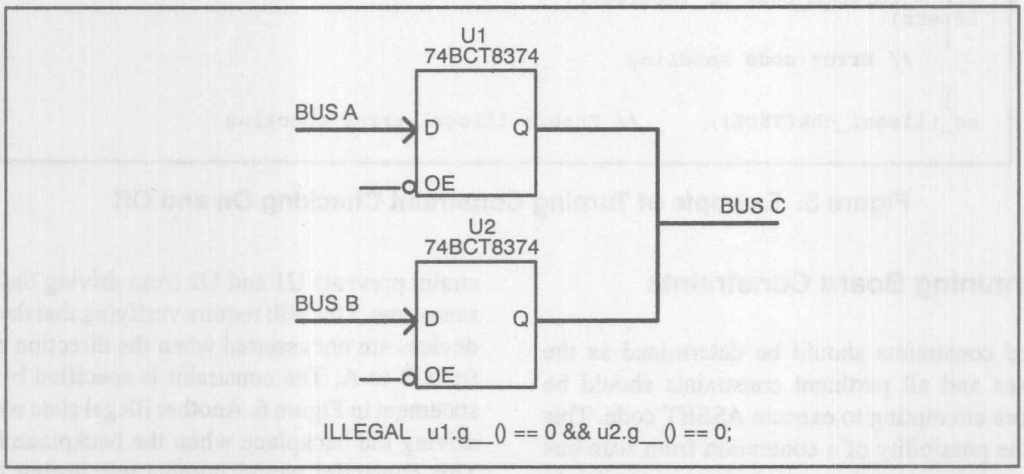


Figure 7. Register Outputs on a Common Bus

The 'BCT8245 in Figure 8 provides partitioning between memory and the local bus. The scan bus must be prevented from back-driving the memory devices when memory is being read. The memory-select lines and the read/write signal on the board should be monitored to prevent contention. The ILLEGAL statement verifies the read-control signal is not being asserted when the direction of U5 is set from A to B.

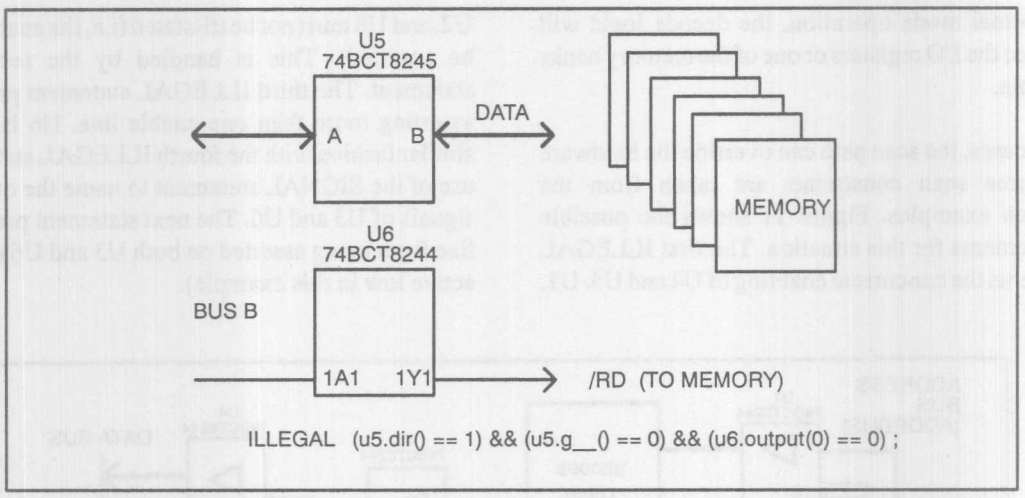


Figure 8. Memory Data Buffer Example

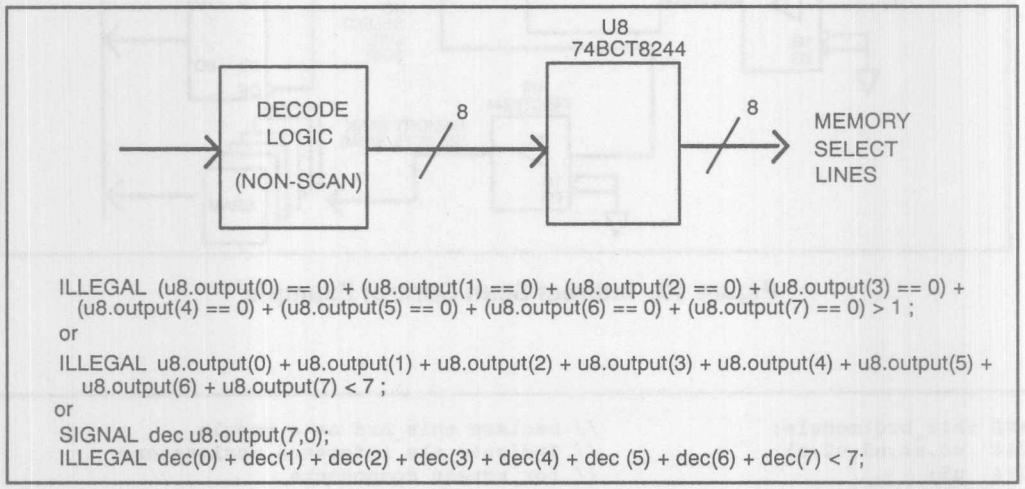


Figure 9. Decode Logic Scan

Figure 9 demonstrates the use of 'BCT8244 octal buffers for buffering decode logic outputs. When the octal outputs are driven by the scan bus, no more than one octal decoding signal should be asserted (shown active low in Figure 9). Asserting more than one output would concurrently select more than one memory bank, causing contention between selected banks during memory read cycles. Three ILLEGAL statement implementations are given for verifying the assertion of only one select line.

Constraint Considerations for Circuits With Indirect Scan Control

The previous examples of scan constraints have been straightforward since they were identical to the hardware design constraints. However, when these circuit configurations are combined on a board, additional scan constraints are required. This is especially true when nonscan circuitry is indirectly controlled via scanning (nonscan circuitry is controlled by scannable parts). Consider the example in Figure 10. This example is a combination of some of the cases discussed earlier. An address bus, either from a backplane or from a processor on the board, is buffered by U1 and U2, which also provides scan access to the address bus. These address lines drive the decode logic, which is composed of nonscannable circuitry. The I/O and memory bank select lines are buffered by U3 and U6, respectively. U4 and U5 represent I/O circuitry. U4 buffers the output of circuitry that is not shown. U5 holds information from other circuitry that is also not shown in the

diagram. In normal mode operation, the decode logic will only select one of the I/O registers or one of the memory banks at any given time.

As in the other cases, the scan path can override the hardware constraints. Some scan constraints are taken from the previously given examples. Figure 11 shows the possible ILLEGAL statements for this situation. The first ILLEGAL statement prevents the concurrent enabling of U4 and U5. U1, U2, and U6 must not be tri-stated (i.e, the enables must always be asserted). This is handled by the second ILLEGAL statement. The third ILLEGAL statement prevents U3 from asserting more than one enable line. U6 is controlled in a similar fashion with the fourth ILLEGAL statement. Note the use of the SIGNAL statement to name the outputs and input signals of U3 and U6. The next statement prevents an enable line from being asserted on both U3 and U6 (enable lines are active low in this example).

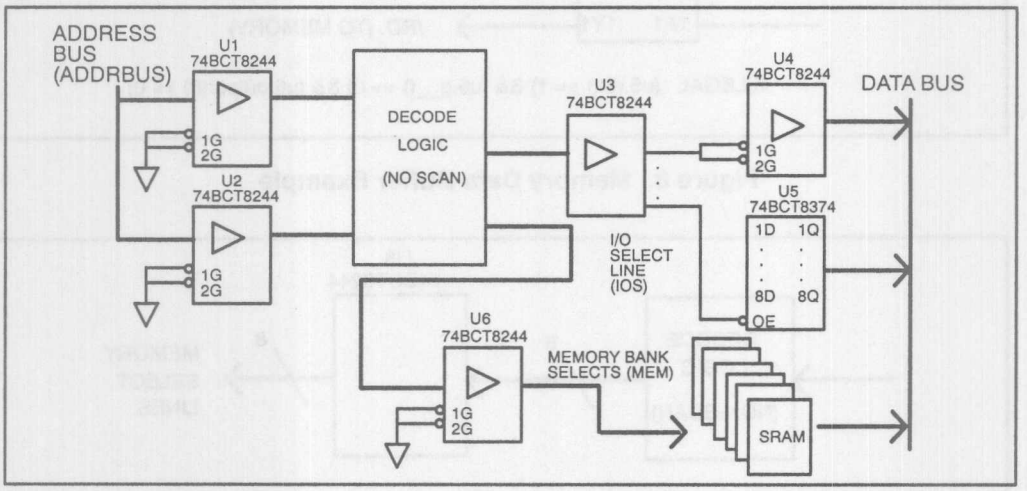


Figure 10. Indirect Scan Control Example

```
CLASS this_brd:module;                  // Declare this_brd as a module
T8244  u6,u4,u3,u2,u1;                  // Indicate the reference designators
T8374  u5;                              // for screen components
MODPATH bpath u6,u5,u4,u3,u2,u1;        // Define scan path order
SIGNAL addrbus u2.output(7,0);          // Assign signal
SIGNAL mem u6.ouput(7,0);               // names for more
SIGNAL decode u6.input(7,0);            // concise ILLEGAL
SIGNAL ios u3.output(7,0);              // statement
SIGNAL iodecode u3.input(7,0);          // definition
ILLEGAL (u4.aux() < 3) && (u5.g__() == 0);
ILLEGAL (u1.aux() != 0) || (u2.aux != 0) || (u6.aux != 0);
ILLEGAL (ios(0) + ios(1) + ios(2) + ios(3) + ios(4) + ios(5) + ios(6) + ios(7)) < 7;
ILLEGAL (mem(0) + mem(1) + mem(2) + mem(3) + mem(4) + mem(5) + mem(6) + mem(7)) < 7;
ILLEGAL (mem() != 0xFF) && (ios() != 0xFF);
ILLEGAL (addrbus() <= 0x03FF) && (u4.aux() < 3)  || (u5.g__() == 0));
ILLEGAL (addrbus() <= 0x3ff) && (ios() < 0xFF);
ILLEGAL (addrbus() == 0x0400) || (addrbus() == 0x0420)) && (mem() < 0xFF);
```

Figure 11. Configuration File for the Example in Figure 10

The remaining ILLEGAL statements prevent the I/O devices from being asserted during memory addressing and vice-versa. Checking U1 and U2 is important since these outputs control the decode logic; therefore, the address driven by these octals must be a scan constraint condition. U4 and U5 are enabled by addresses 0400 (hex) and 0420, respectively, and SRAM is addressed in the range of 0 to 03FF. If U1 and U2 are driving the address of 0100 (hex), for example, then neither U4 nor U5 are permitted on the bus. If U1 and U2 are driving address 0400, then U5 cannot be asserted on the bus without changing the address on U1 and U2. It can be seen from this example that indirect control of circuitry by scan devices can complicate the determination of scan constraints.

Constraint Determination Example

An example of a memory board designed with scope octals is given by Figure 12. It is designed to interface to the VME bus as a slave module. The buffers normally used at the connector are replaced by TI SCOPE octal components. The reference designations for the scan components of interest are given in Table 2. Several of the constraints used in previous examples are applicable to this design. Figure 13 shows the applicable ILLEGAL statements. An obvious constraint situation involves the two sets of data transceivers. These are handled by the second and third ILLEGAL statements in Figure 13. As before, the scan constraint must prevent the transceivers from driving the local data bus at the same time. U96 and U98 are the octal data components at the connector, and U60 and U77 are the octal components between the local data bus and the memory data bus.

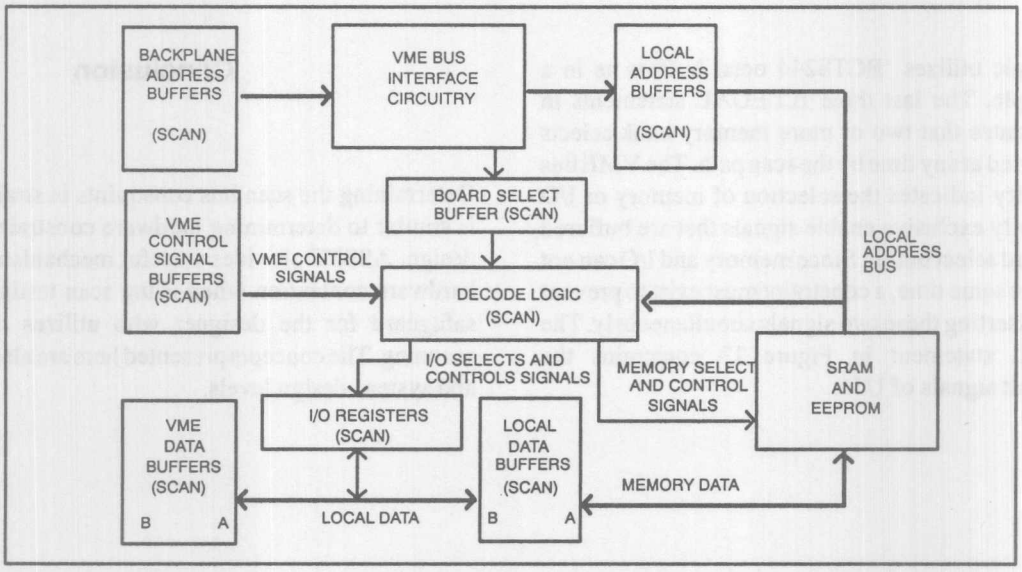


Figure 12. VME Memory Board

Table 2. Partial List of Scan Devices for Memory Board Example

Reference Designator	Type of Component	Circuit Block
U96	74BCT8245	VME Data Buffer
U98	74BCT8245	VME Data Buffer
U60	74BCT8245	Local Data Buffer
U70	74BCT8245	Local Data Buffer
U37	74BCT8244	Decode Logic
U38	74BCT8244	Decode Logic
U51	74BCT8244	Board Select Buffer

```
SIGNAL mem1 u38.output(7,0);    // Assign names to device outputs
SIGNAL mem2 u37.output(7,0);
ILLEGAL !u51.output(4) && !u51.output(3);

// Prevent u96 and u60  from backdriving each other
ILLEGAL u96.dir() == 1 && u96.g__() == 0 && u60.dir() == 0 && u60.g__() == 0;

// Prevent u98 and u77 from backdriving each other
ILLEGAL u98.dir() == 0 && u98.g__() == 0 && u77.dir() == 0 && u77.g__() == 0;

// Do not allow u116 ('BCT8245) to drive from B to A
ILLEGAL u116.dir() == 0 && u116.g__() == 0;

//Do not permit assertion of more than one device select signal
ILLEGAL (mem1(0) + mem1(1) + mem1(2) + mem1(3) + mem1(4) + mem1(5) +
         mem1(6) + mem1(7)) < 7;
ILLEGAL (mem2(0) + mem2(1) + mem2(2) + mem2(3) + mem2(4) + mem2(5) +
         mem2(6) + mem2(7)) < 7;
ILLEGAL (mem1() != 0xFF) && (mem2() != 0xFF);
```

Figure 13. Constraints for Memory Board Example

The decode logic utilizes 'BCT8244 octal buffers as in a previous example. The last three ILLEGAL statements in Figure 13 guarantee that two or more memory bank selects will not be asserted at any time by the scan path. The VME bus interface circuitry indicates the selection of memory or I/O with two mutually exclusive enable signals that are buffered by U51, the board select buffer. Since memory and I/O can not be selected at the same time, a constraint must exist to prevent the scan from asserting these two signals simultaneously. The first ILLEGAL statement in Figure 13 constrains the respective output signals of U51.

Conclusion

Determining the scan bus constraints is straightforward and is similar to determining hardware constraints during board design. ASSET provides a useful mechanism for preventing hardware contention when using scan testing. It provides a safeguard for the designer who utilizes designed-in test scanning. The concepts presented here are also valid at the IC- and system-design levels.

"What's an LFSR?"

by John Koeter

Introduction

The purpose of this article is to explain what a Linear Feedback Shift Register (LFSR) and a Parallel Signature Analyzer (PSA) are and how to apply them to test a TI ASIC (Application-Specific Integrated Circuit) using SCOPE cells. This article starts off with a description of an LFSR, goes into Pseudo-Random Pattern Generation (PRPG) and fault grading, describes a PSA, and then shows how to implement the PSA and LFSR functions using SCOPE boundary scan cells, which are compatible with the IEEE 1149.1 standard.

LFSR

An LFSR is a shift register (a function that advances the signal from one bit to the next most-significant bit when clocked. See Figure 1) with some of the outputs exclusively ORed together to form a feedback mechanism.

A linear feedback shift register can be formed by EXORing the outputs of two or more of the flip-flops together and feeding them back into the input of one of the flip-flops as shown in Figure 2.

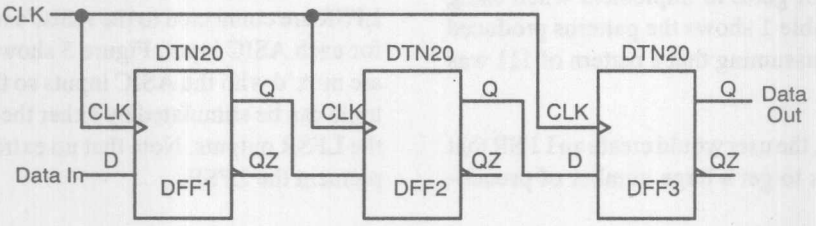


Figure 1. A 3-Bit Shift Register

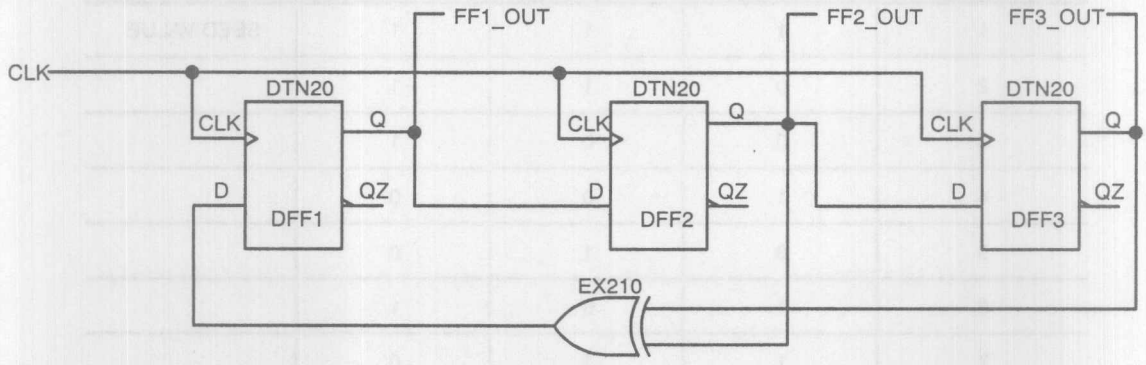


Figure 2. Linear Feedback Shift Register

Pseudo-Random Pattern Generation

Linear feedback shift registers make extremely good pseudo-random pattern generators. When the outputs of the flip-flops are loaded with a seed value (anything except all 0s, which would cause the LFSR to produce all 0 patterns) and when the LFSR is clocked, it will put out a pseudo-random pattern of 1s and 0s. Note that the only signal necessary to generate the test patterns is the clock.

Maximal Length LFSRs

A maximal length LFSR produces the maximum number of PRPG patterns possible and has a pattern count equal to 2^{n-1}, where n is the number of register elements in the LFSR. It produces patterns that have an approximately equal number of 1s and 0s and have an equal number of runs of 1s and 0s.[1]

Since there is no way to mathematically predict if an LFSR will be maximal length, Peterson and Weldon[2] have compiled tables of maximal length LFSRs to which designers may refer. These tables have been reproduced in the *Scope Cell Design Manual*[3] along with a listing of maximal length LFSRs that require the least number of gates to implement when using TI's ASIC Scope cells. Table 1 shows the patterns produced by the LFSR in Figure 2, assuming that a pattern of 111 was used as a seed.

In a practical ASIC design, the user would create an LFSR that is much bigger than 3 bits to get a large number of pseudo-random patterns before the patterns repeated. There are some practical restrictions to the length of the LFSR however. A 32-bit maximal length LFSR would create over 4 billion patterns that, assuming a 16-MHz clock rate, would take almost 5 minutes to generate the whole pattern set.

PRPG and Fault Grading

The LFSR and the Pseudo-Random Pattern Generation technique is often used to create functional patterns that will provide a high level of fault coverage for the ASIC with minimum effort by the designer or the test engineer. Pseudo-randomly-generated patterns have been proven to very quickly generate high fault coverage results. The PRPG technique works especially well for combinational logic but may also work well for certain cases of sequential circuits because each input signal stimulated by the LFSR is frequently changing from a one to zero and back again (high bit toggle rate).

A designer would design the ASIC circuit as normal and simulate the design to verify correct functionality and timing. Once verified, the LFSR is designed and the outputs of the LFSR are connected to the ASIC's inputs – one LFSR output for each ASIC input. Figure 3 shows how the LFSR outputs are mux'd with the ASIC inputs so that the ASIC application logic can be stimulated by either the normal data inputs or by the LFSR outputs. Note that no extra pins are required to implement the LFSR.

Table 1. Pattern Generator Seed Values

CLOCK PULSE	FF1_OUT	FF2_OUT	FF3_OUT	COMMENTS
1	1	1	1	SEED VALUE
2	0	1	1	
3	0	0	1	
4	1	0	0	
5	0	1	0	
6	1	0	1	
7	1	1	0	
8	1	1	1	STARTS REPEAT

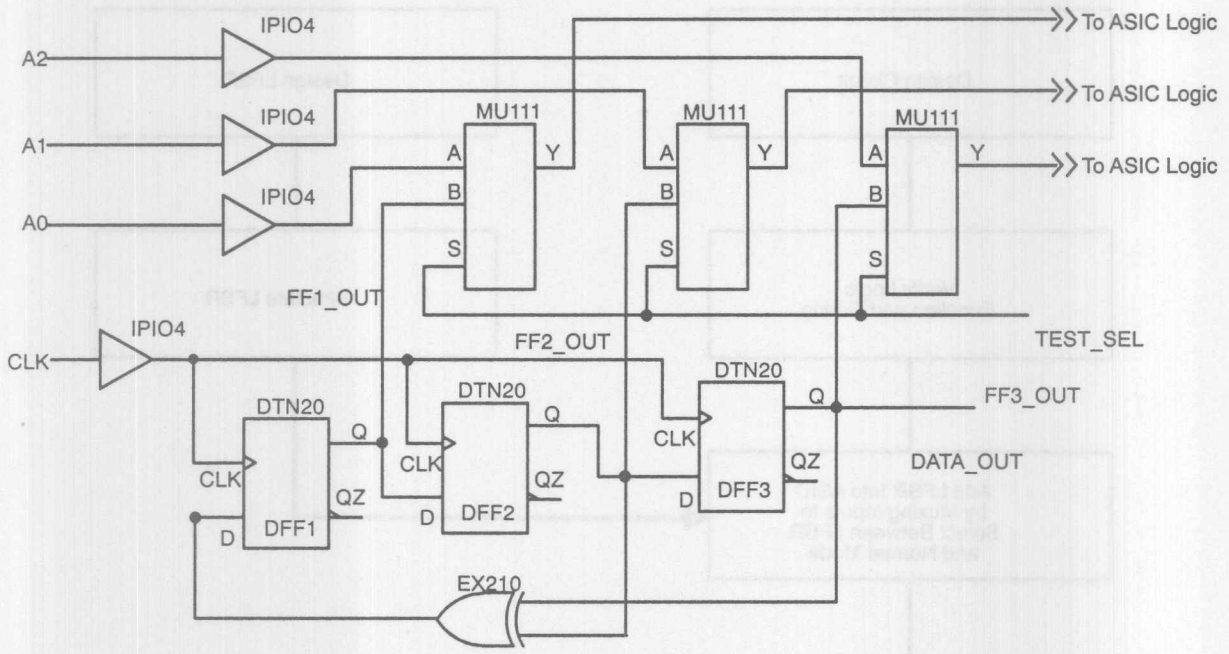


Figure 3. LFSR With Outputs Multiplexed with ASIC Inputs

A designer would typically capture the LFSR (or PSA) and simulate it by itself to evaluate and verify this subcircuit without having to simulate the total design. The designer could then quickly examine the effects of different seed values on the patterns produced. This is particularly important if not all the 2^{n-1} patterns that the LFSR could produce will be used in circuit testing. A designer would typically print out a sample of the patterns (every tenth or hundredth pattern, etc.) and use the printout when simulating the LFSR in the complete circuit to verify that the LFSR is generating the correct signals. Since the logic of the circuit has already been verified, the device would be put in the test mode to let the LFSR generate the inputs to the ASIC. Next, the outputs are sampled every clock cycle to generate the expected outputs given the inputs that the LFSR has created.

Once the designer has verified that the LFSR is generating the correct circuit inputs, he/she would use the resulting simulation vectors to fault grade the design. Fault grading enables the customer to ensure that the test vector set supplied for the design will catch manufacturing defects such as shorted transistors and open metal lines (stuck-at faults, SA0, SA1). If the fault grade that results from applying the LFSR generated simulation patterns is lower than required, then the designer would have to generate additional patterns to raise the fault grade level to the required value.

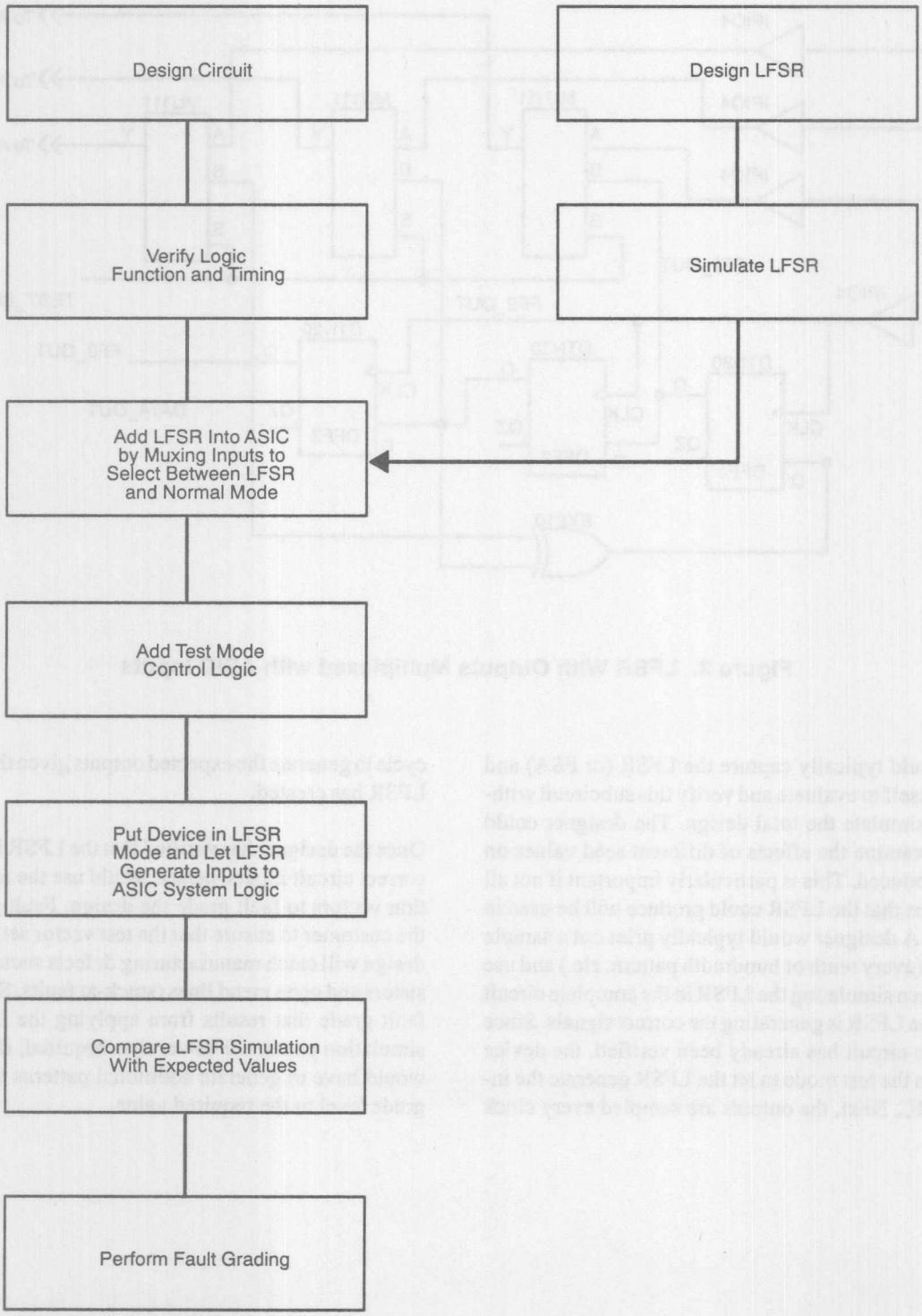


Figure 4. Flowchart for Designing an LFSR Into an ASIC

External LFSRs

While the above section focused on how to use the LFSR for Built-In Test (BIT) of an ASIC, it should be noted that the LFSR logic incorporated in an ASIC can also be used to drive external logic. This is especially easy to do if TI's SCOPE cells have been used to incorporate boundary scan into the design. (See the SCOPE and LFSRs section.) In this scenario, the LFSR would be mux'd with the ASIC outputs so that when the device is placed in a test mode, pseudo-random patterns would be generated and applied to the board logic. Board faults could then be detected by monitoring the board's edge connector or by capturing the logic's outputs and scanning the information out through the 1149.1 test bus.

Pattern-Resistant Logic

A major potential problem with the LFSR approach is that some logic is pattern resistant. Take for example an NA810 (an 8-input NAND gate). When any of the 8 inputs is a 0, the output is a 1. Only when all eight inputs are 1s, does the logic change state. A manufacturing defect like a stuck at 0 fault on the output of the NA810 will be extremely hard to detect using random patterns. Another problem is some control signals should not toggle as often as they would when stimulated by an LFSR. A system reset or a clear pin is a good example. If this were toggling all the time, the logic would keep resetting and you probably wouldn't get the best fault detection result. Sequential logic circuits also provide a problem. For example, if state 3 in a state machine can only be achieved if the device first enters into state 1 and then state 2, a random pattern sequence may have difficulty in creating inputs that move the circuit from one state to another in the correct order.

These problems are solved by a variety of techniques including using longer LFSRs to generate virtually every possible input, holding key control signals to a fixed value during the test stage, and augmenting the patterns with hand generated patterns to improve fault grading.

PSA

As was mentioned above, an LFSR of any significant length will generate a large number of patterns. To avoid having to test the outputs of several hundred thousand or more vectors, a Parallel Signal Analyzer (PSA) is used to compress the data at the outputs of the ASIC. As Figure 5 indicates, the PSA is nothing more than an LFSR with EXOR gates between the shift register elements. In fact, a PSA can be used as an LFSR if the A_IN, B_IN, and C_IN inputs are all held at 0. If the inputs are held at 0, the PSA will generate exactly the same patterns as the LFSR in Figure 2.

Historically however, the PSA is most often used as a parallel to serial compression circuit. The A_IN, B_IN, etc. inputs are mux'd with the ASIC outputs. As each pattern is applied to the ASIC by the LFSR connected to the ASIC inputs, the output state of the ASIC is read into the PSA. As each new pattern is applied, the PSA will exclusive-OR the last pattern's outputs with the current pattern's output to create a new value in the PSA. This conceptually is very similar to a calculator adding a series of numbers. For example, if you are adding 2 + 3 + 6 + 9 + 1 + 1 using a calculator you would first add the first state, 2, to the second state, 3 to get 5. You would then add the new state, 6, to the old state, 5, to get the new result 11, etc. Instead of using addition, the PSA exclusive-ORs the series of 1's and 0's together to get the new result.

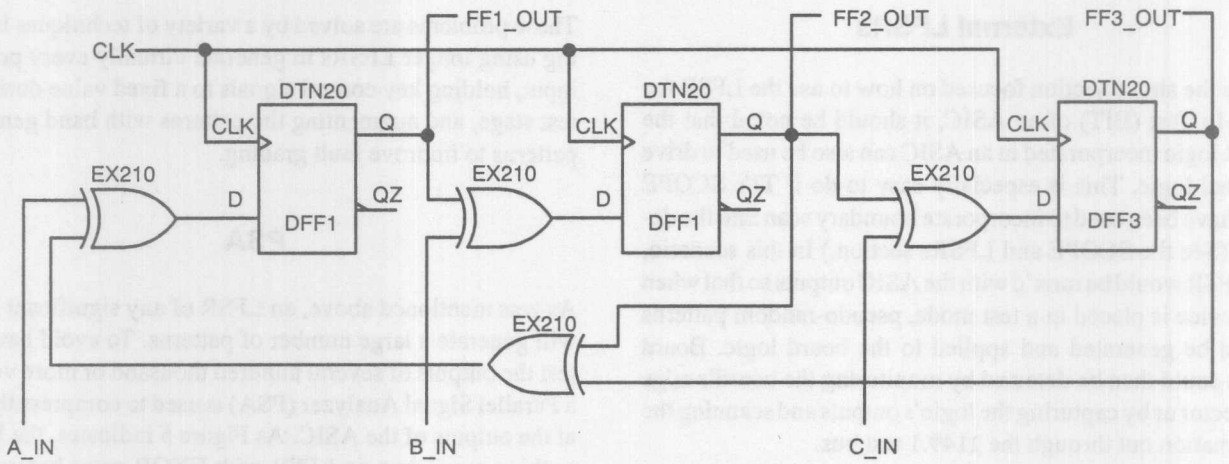


Figure 5. A Parallel Signal Analyzer

Table 2. PSA Signatures

INPUTS					OUTPUTS		
Clock Pulse	A_IN	B_IN	C_IN		FF1_OUT	FF2_OUT	FF3_OUT
				Seed Value	1	0	0
1	1	1	1		1	0	1
2	0	1	1		1	0	1
3	0	0	1		1	1	1
4	1	0	0		1	1	1
5	0	1	0		0	0	1
6	1	0	1		0	0	1
7	1	1	0		0	1	0
8	1	1	1		0	1	0
Final Signature = Pattern 8 Output Result = 010							

After a predefined number of patterns, the results in the PSA are read out of the ASIC (via FF1_OUT, FF2_OUT, and FF3_OUT) and compared to the expected value. Table 2 shows the signature that would be read out of the PSA given that the inputs to the PSA were the outputs of the LFSR in Table1.

Aliasing

One major problem in compressing the results of a number of patterns into one pattern, called a signature, is aliasing. Aliasing occurs when the signature of the PSA is the same as expected but it was caused by a cancellation of errors in the patterns. Thus the silicon fault that caused the incorrect circuit response to the input pattern is never detected. To use the calculator example, 2 + 3 + 6 + 9 + 1 + 1 = 22, but so does 2 + 2 + 7 + 9 + 1 + 1. In this example, the second and third value that represent incorrect circuit values cancel each other out. Note that aliasing can only occur when there are two or more incorrect output patterns.

A way to minimize the aliasing problem is use maximal length PSAs and to more frequently read out the PSA signature for comparison to a known value or to monitor the most-significant PSA bit (FF3_out in this case) continuously.

SCOPE, LFSRs, and PSAs

Boundary scan is a test methodology whereby information/instructions can be scanned through a device, board, or system and monitored/executed. It provides controllability and observability of internal nodes and may reduce or even eliminate the need for a conventional bed of nails tester.

SCOPE is TI's proprietary name for its boundary-scan products. SCOPE is 100 percent compatible with the IEEE 1149.1 boundary-scan standard but offers an extended instruction set and more capabilities than the basic standard.

Versions of the SCOPE boundary scan cells that TI has in the TGC100, TSC500, and TSC700 libraries contain the logic necessary to implement LFSRs and PSAs with a minimum of work and a minimum of logic added by the designer. To implement boundary scan, the design would need to have a TSG00 cell on every input and every output. The TSG00 cell, which is used to implement the boundary scan register, is basically a shift register element with a mux connected to the ASIC input that selects between normal and test mode. Thus with the additional of exclusive-OR gates, the TSG00 can be configured as a linear feedback shift register and/or a parallel signature analyzer element. By incorporating the exclusive-OR gates, the ASIC will have LFSR and/or PSA capabilities and be fully compatible with the IEEE 1149.1 standard.

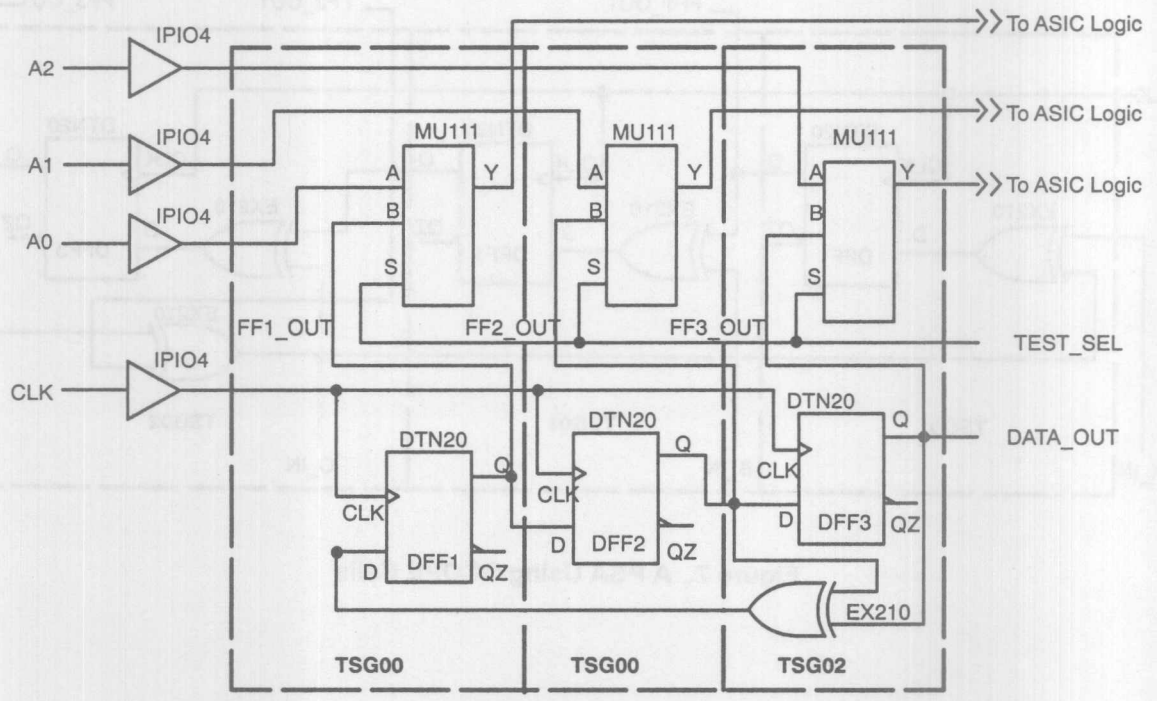


Figure 6. An LFSR Using SCOPE Cells

The TSG01 and TSG02 cells have the necessary EXOR logic embedded in them to provide the feedback. For the LFSR, the shift register elements that have feedback taps (FF3 in Figure 2) would use the TSG02 cell. The FF1 and FF2 shift register would still use the TSG00 cell. Figure 6 shows the LFSR of Figure 2 implemented using SCOPE cells.

Similarly, the PSA can be implemented easily with the few extra gates of logic that are contained in TSG01. To configure a PSA using SCOPE cells, the TSG01 cell would replace the TSG00 (FF1 and FF2 in Figure 5) and TSG02 cell would be used on the pins that needed feedback (FF3 in Figure 5). Figure 7 shows the PSA of Figure 5 implemented using SCOPE cells.

The SCOPE cell design manual provides good examples of how to design circuits using these cells. Note that TSB00, TSB01, and TSB02 cells would be used in place of TSG00, TSG01, and TSG02 cells if the pins were bidirectionals.

If the boundary-scan register will be configured as an LFSR or PSA, a boundary-control register must be added to the design. A boundary-control register contains the additional signals necessary to reconfigure the boundary-scan register as an LFSR or PSA. SCOPE cell instructions have been defined that, when decoded by the boundary-control register, will provide the control signals necessary to run built-in-tests using LFSRs and/or PSAs. The SCOPE cell design manual contains a full description of the relevant instructions and the control signals that the boundary-control register provides to the boundary-scan cells.

In a typical application, the customer would assemble the boards and test the boundary-scan ring to verify the scan path. He/she would then initialize the ASIC to a known value and put the ASIC in a boundary-scan mode. The RUNBIST instruction would be scanned into the instruction register (via TDI) and decoded via the boundary control register. The control signals thus generated would configure the appropriate input boundary scan cells into an LFSR and the appropriate output boundary scan cells into a PSA. The LFSR would have been "seeded" with a known value during the initialization phase and thus once clocked (by TCK) would start generating input stimulus. The PSA would capture the output state of the circuit and compress the results. After a given number of TCK cycles, the signature would be scanned out via TDO to be compared with a know value.

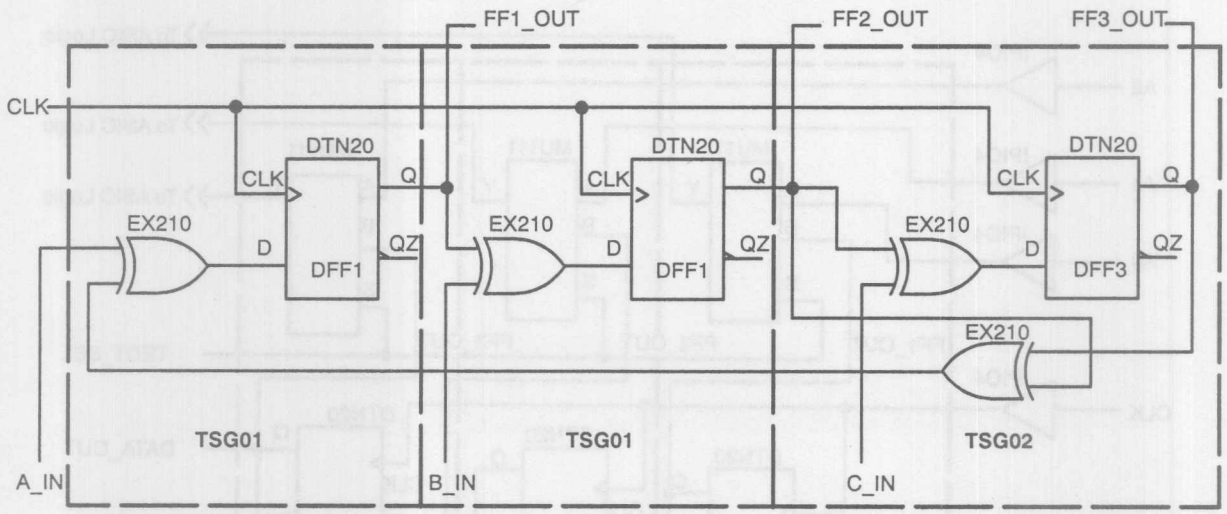


Figure 7. A PSA Using SCOPE Cells

Summary

The LFSR is a shift register that has some of its outputs exclusively ORed together to form a feedback path. LFSRs are frequently used as pseudo-random pattern generators to generate a random number of 1s and 0s. Each output of the LFSR is mux'd with an ASIC input and when the device placed in the LFSR (test) mode, the random, high toggle rate patterns produced are extremely good for generating high fault coverage. To minimize the number of results that need to be compared to expected results, a parallel signature analyzer is used. The PSA compresses multiple parallel patterns into a single pattern signature that is compared to the expected value. If the signatures match, it is assumed that the ASIC passed the test vectors applied and there are no manufacturing defects.

Definitions

Sequential logic – logic functions whose next state depend upon both the inputs to the function and its current state.

Combinatorial logic – logic functions that depend only on the inputs to the function.

Pattern resistant logic – logic for which the pseudo-random pattern generation technique is not well suited because the random patterns will not cause a stuck-at fault to be caught.

Signature – the final result of the compression of a number of patterns using a PSA.

Aliasing – when the signature of a PSA is correct because two or more errors cancelled each other.

Stuck-at faults – manufacturing defects (such as a shorted transistor or an open metal line) that cause an input or an output of a logic function to be permanently stuck at a high or low logic level.

References

[1] *Self-Test Services*, The STS ASIC Testability Seminar, Self-Test Services, Ambler, PA, 1989.

[2] W.W. Peterson and E.J. Weldon, Jr. *Error Correcting Codes*, MIT press, Cambridge, MA 1972.

[3] Texas Instruments, *SCOPE Cell Design Manual*, Texas Instruments, Dallas, TX, 1990.

Writing ASSET Subroutines to Aid In Debug and Test

By Victoria Gobeli

Introduction

ASSET (Advanced Support System for Emulation and Test) subroutines can be used for scan-based testing and diagnostics to verify the hardware design of hardware. This application note will explain how to develop subroutines to aid in the debugging and testing of hardware designs. The major topics include: how to identify the subroutines that need to be written in order to test all functional areas of the design, how to construct these subroutines, how variables are passed into these routines, user interfaces, source code compiling information, and testing/verification of written subroutines.

Overview of ASSET Subroutines Writing

ASSET subroutines can be constructed easily when certain methods are used. The first step is determining which functional areas of the hardware design need to be tested. For example, functional areas of a memory board requiring testing may include RAM, EEPROM, local registers, read/write control signals and memory select lines. Once these areas have been determined, a subroutine can be written to test each area, such as writing/reading data to and from RAM, or writing control patterns to registers. Input parameters must be determined for each subroutine. These parameters may include an address and a data word for writing to RAM, or a control pattern and register address for writing registers. When the input parameters have been determined, output parameters must be established in order to verify the correct functional operation of the design. Interfaces must be established to allow the user to pass parameters to the routines and to pass the results back to the user in a readable, understandable, and easy to use format. Compiling the subroutines is the next phase of ASSET subroutine writing. All subroutines should exist in the same file for modularity and ease of compiling. Once the subroutines have been written and compiled, they should be fully tested for possible software and hardware design errors.

Determining Which Subroutines to Write

Partitioning Functional Areas of Hardware Design

To test a hardware design easily, the design needs to be separated into distinct functional test areas. As an example: a memory board's partitioned functional areas could include static ram, dynamic RAM, EEPROM, and any local registers. Control signals such as memory chip selects, read and write strobes and reset lines, can also be tested. This separation will also help to isolate any hardware design errors by narrowing them down to a specific functional area. An example is data written to static RAM and then read back. If the data pattern read varies only by a toggled bit from the data pattern written, it can be suspected that a data line has a stuck-at-one or a stuck-at-zero fault. On the other hand, if the data pattern read varies greatly from the data pattern written, a memory-select line could be wired incorrectly, enabling EEPROM rather than static RAM.

Establishing Functional Operations for Each Test Area

After each functional area has been partitioned, all the functional operations for that particular area need to be established. For a static RAM area on a memory board, both read and write operations can be performed. For local registers, control patterns can be written to verify, for example, that certain LEDs light up. Basically, a list of all hardware functional operations for a particular area is made, and a subroutine is developed for each operation. Functional operations can be extracted from the design specification documents for the hardware design and by viewing the schematic drawings.

Map Subroutines to Each Functional Area

A subroutine should be written to test each operation for each functional area of the hardware design. For easy determination, the name of the subroutine should match the operation being performed, such as Read_SRAM or Write_SRAM. This will also allow the user to write modular and less complicated code. The fewer functional operations being performed per subroutine, the easier each subroutine is to test and verify, and the easier it is to isolate a fault down to a particular functional area.

Constructing ASSET Subroutines

Developing Pseudo Code

The first step to writing any subroutine is to develop pseudo code for each subroutine. Pseudo code consists of simple English sentences that trace a step-by-step flow for executing a particular function. This flow allows the user to visualize the function being performed. A memory board is a good example. A functional area to be tested would be SRAM, and an operation performed in this area would be reading a word of memory. Pseudo code for this operation might look like:

a. Initialize control lines
b. Initialize select lines
c. Enable data memory devices
d. Drive address
e. Assert address strobe
f. Assert read strobe
g. Wait for acknowledge
h. Read data
i. De-assert read strobe
j. De-assert address strobe

Pseudo code such as this should be mapped out for each operation of each functional area to be tested.

Determining Input/Output Parameters

Once the pseudo code has been developed, the input and output parameters need to be determined. Input parameters are used to make the tests more flexible. Instead of writing to the same address each time for a memory write, it is more useful to allow the user to select any address by passing it in as an input parameter into the subroutine. When writing data patterns, it is more useful to write user-selected patterns, rather than hard coded data values. Especially if the user is trying to debug stuck-at faults, specific data patterns need to be applied depending on which suspected data bit is stuck. Output parameters are used to pass results of the functional operation back to the user. When performing read operations, the user needs to know what the data value read is. Or when performing write operations, the user may need to know which memory-bank line was activated for the given address.

Determining input/output parameters can be accomplished simply by examining the pseudo code and determining which information needs to be supplied by the user. Looking at the above pseudo-code example, control lines and select lines are always set to some predetermined value, active-high or active-low. However, the address that is driven out is unknown. The address and read strobes have some predetermined value, but the data read back is unknown. An input parameter for this operation could be an address, and an output parameter could be the data value read back. The read operation can be broken down further into reading words or bytes. In this case, another input parameter would be needed to determine the size of data being read, whether 8 or 16 bits.

Developing C++ Code Based on Pseudo Code

C++ code can now be developed from the pseudo code. Each pseudo-code sentence needs to be further broken down into actual functional operations. Once each functional operation is determined and the devices needed to perform this operation are known, the ASSET C++ code can be written. Initializing control lines for a read operation entails determining which control lines need to be initialized and from what octal devices they are driven. With the appropriate ASSET functions, each octal device is set into a test mode, the initialized value is scanned into the device, then the value is driven out of the octal, initializing each control line to a predetermined value. This process is performed for each octal that comprises the pseudo-code operations for a functional area.

User Interfaces

The user interfaces are screen-display interfaces between the user and the subroutines. The input and output parameters to the subroutines need to be passed in a structured and readable format. For example, when using digital octal data devices, it is more convenient to display data values as hexadecimal numbers rather than as a long decimal number. Menus and sub-menus can be constructed according to the functional areas of the hardware design being tested. For example, one menu could be labeled "SRAM" with further sub-menus defined underneath labeled "Read SRAM Word", "Write SRAM Word", "Read SRAM Byte", etc. Each menu should contain functionally equivalent operations for easy partition-

ing. Group all SRAM operations together, or group all read operations together in the same menu, etc.

Input Parameters

Input parameters passed into the subroutines can be supplied by the user from simple prompts asking the user for responses. Prompting the user for an address in a read memory subroutine will supply the input parameter to the subroutine. These prompts can be written using functions supplied with the basic C++ programming libraries.

Output Parameters

The output parameters from each subroutine need to be displayed to the user in a readable and understandable format. An example would be the data read back from a read memory subroutine. The display should include a title that is descriptive for the item being displayed and a pause capability so that the user can view the item as long as needed, then continue on by pressing any key. These display capabilities are provided with the standard C++ library functions.

Source Code Compiling

Compilation of subroutine source code is the next phase of writing ASSET subroutines. Following C++ programming guidelines, each subroutine must exist within the predefined class type for the hardware design. This will guarantee that the functional operations defined in each subroutine for a particular hardware module will only be performed on that one module in an entire system of modules. Following the ASSET programming guidelines, a configuration file should have already been established that defines the octal types, scan paths, and any signals used by the subroutines. During compilation, the class type and configuration file will be accessed to verify correct manipulations of the octals that are used to perform each functional operation.

Resolving C++ ASSET Errors

Any number of errors could occur during compilation, ranging from programming syntax errors to inoperable functions performed on a specific octal. Most errors can be resolved by referring to the Zortech C++ Compiler User's Guide. Here are a few hints to minimize the error debug process. Each module has a defined configuration file associated with it (these configuration files are described in detail in a separate application note). Defined within this configuration file is an 'elements.add' structure that identifies all octal devices on the module. Also in the configuration file is a 'signals.add' structure that identifies all user-defined signals for the module. Verify that all elements and signals defined in the 'elements.add' and the 'signals.add' match the number prefaced in the command statement. For example: signals.add (4, &sig1, &sig2, &sig3, &sig4);

The total number of signals defined is four, which matches the number prefaced at the beginning of the statement. Also make sure that the '&' character appears in front of each signal definition. Erratic results will occur at run time if these statements are formatted incorrectly. Signals can be used to simplify programming of ASSET subroutines. A signal can be defined to represent a 32-bit bus spread over four separate octal devices. This eliminates the need to mask and shift all the bits of each octal device into one representative number. The following is an example of a signal definition for a data bus:

ram_data ("ram_data", 4, &u1.input,7,0, &u2.input,7,0, &u3.input,7,0, &u4.input,7,0);

The "ram_data" signal comprises octals U1, U2, U3 and U4 and utilizes the input pins 0–7 of each device.

Each subroutine written must be included in the class definition for the hardware module defined. This allows the subroutines to access all octals on the board that are needed to perform the specific functional operations. The following example defines a read and write subroutine for a memory board.

Header file defined for memory board:

```
class MEM_BOARD:public module { public:
MEM_BOARD(char *id);

T8244 U1, U2; T8373 U3, U4;

void read_SRAM (unsigned long address, unsigned long *data);

void write_SRAM (unsigned long address, unsigned long data); }
```

Source code file for subroutines:

```
MEM_BOARD::read_SRAM (unsigned long address, unsigned long *data) { // source code exists here }

MEM_BOARD::write_SRAM (unsigned long address, unsigned long data) { // source code exists here }
```

In the above example, class type MEM_BOARD has been defined. Located on the Memory board are octal devices U1, U2, U3 and U4. U1 and U2 are defined to be octal buffers. U3 and U4 are defined to be octal D-type latches. Therefore, subroutines read_SRAM and write_SRAM can only be performed on octal devices U1, U2, U3 and U4 for the Memory board MEM_BOARD.

Subroutine Testing and Verification

When the subroutines are executed, both the software and hardware designs will be verified. Once the software bugs are eliminated, the subroutines can be used on all similar hardware designs to verify correct hardware functional operation.

Software Verification

If a subroutine returns an unexpected value, either the software or the hardware is at fault. Re-examine the pseudo code to make sure that the correct operations are being performed in the correct order for that specific function. Timing could also be an issue, so verify that all strobes are being asserted at the appropriate time. Also make sure that all octals utilized in the operation are enabled, driving out in the correct direction and driving out valid data. When the subroutine has completed its functional operation, all octals used in the operation must be returned to their original functional state.

The ASSET debugger is also available to aid in source code debugging. (The ASSET debugger is described in detail in a separate application note.) The debugger can be used to single-step and trace through the source code line by line for easily identifying the software bug.

Hardware Verification

Once the software design has been fully debugged, the hardware design can be verified. Diagnostic equipment such as logic analyzers and oscilloscopes will facilitate the hardware debugging process. If a sub-routine returns an unexpected value, the hardware needs to be probed further for problems. The ASSET debugger can also be used to aid in the debug of hardware designs by examining the current state of octal devices, or initializing them to some predefined state.

Conclusions

Debugging hardware designs using ASSET subroutines can be easily achieved. All that is needed is a complete working knowledge of the design being tested, an understanding of the operations performed by each functional area of the design, and a familiarity with the C++ programming language and ASSET environment. Once all of the above requirements are met, the debug and test of a hardware design can be simply facilitated and accomplished.

Glossary

A

ASSET: *Advanced Support System for Emulation and Test.* ASSET is an integrated test program and test hardware development tool for use with 1149.1-based systems and components.

ATE: *Automatic Test Equipment.*

Automatic Test Equipment: *See ATE.*

B

BIST: *Built-In Self-Test.* Logic is included in a design that performs a self-contained self-test of the device.

boundary control register: A TI user-defined register required for use with SCOPE products – also known as SCOPE control register.

boundary scan: A testability methodology in which a partitioning scan ring is incorporated at the perimeter of a design to provide ability to control and observe access via scan operations. Inserting a test partition at a device boundary allows access to both the internal (or device) and external (other device) logic. Individual device scan rings may be interconnected to form larger partitions in electronic circuitry.

boundary scan register: An IEEE 1149.1 required register that partitions a section of functional logic to enable the control and observation of the logic.

Built-In Self-Test: *See BIST.*

bypass register: An IEEE 1149.1 required register that abbreviates the scan path length to 1 bit.

C

CAD/CAM: *Computer Aided Design/Computer Aided Manufacturing.*

CLASS: CLASS defines the name of the device or module type. (For example: `CLASS mychip:device;` and `CLASS mybrd:module;`.)

configuration file: This file describes devices and modules used by the ASSET Application Program and is analogous to a CAD Netlist. (See Device Configuration File.)

D

device: A single 1149.1 component (chip).

device configuration file: A description of 1149.1 test logic in terms of individual scan registers, the length of these scan registers, register subcomponents, and the test operations that can be performed on these scan registers, or by them. (Describes a single 1149.1 component (chip).) This information is used by the ASSET application program to build software representations of the hardware components to be tested.

DEVPATH (device path)**:** This statement is used to select the 1149.1 test register to be addressed on an 1149.1 device. Only one register (Instruction Register (IR) or Data Register (DR)) may be specified per devpath.

F

FIELD: A functional partition of an IEEE 1149.1 test register.

functional logic: That portion of the electronic circuitry in a device (or system) associated with the normal functional op-

erations. In 1149.1 devices (or systems), the term 'functional logic' specifically refers to all of the non-1149.1 circuitry within the device (or system).

H

hierarchical testability: A test approach that partitions a system into smaller subsystems.

hierarchy: The test architecture of an IEEE 1149.1-based system expressed in terms of lower level ASSET modules and ASSET devices.

I

IEEE 1149.1: A scan-based architecture for use in test applications. ICs that conform to the 1149.1 standard can be connected by a common four-wire bus. An external 1149.1 bus controller (such as ASSET) can observe and control critical test nodes on the target unit via the scan interface.

IEEE 1149.1 device: A device that does not control the bus on which it resides. An 1149.1 device reacts passively to commands issued by the 1149.1 TBC. For example, TI ASSET octals are 1149.1 devices. Any number of 1149.1 devices may exist on an 1149.1 bus.

IEEE 1149.1 Test Bus Controller (TBC): The module that controls the 1149.1 bus. The module issues commands and scans data in and out of 1149.1 devices. Only one 1149.1 TBC can be active on an 1149.1 bus at any time.

ILLEGAL: An ILLEGAL statement defines constraints that can cause potentially damaging hardware contention if violated. If ILLEGAL state checking is enabled, ASSET evaluates all constraints prior to a DR-Scan.

include files: Files needed to execute an ASSET program.

instruction register: An IEEE 1149.1 register that enables the storage of 1149.1 commands. A 2-bit register is required; SCOPE requires an 8-bit register.

J

JTAG: *Joint Test Action Group.* JTAG was composed of representatives from companies in the electronics industry who sought answers to testability issues. JTAG was instrumental in the effort to get a scan-based architecture accepted as IEEE standard 1149.1.

Joint Test Action Group: *See JTAG.*

L

LFSR: *Linear Feedback Shift Register.* A shift register counter that can be used for pseudo-random test pattern generation and for compression of test pattern results.

library: A directory containing one or more object module files.

Linear Feedback Shift Register: *See LFSR.*

M

menu: A software screen that allows selection from among several options.

MODPATH: Defines one scan path on the module and a list of the 1149.1 devices or modules contained in that scan path.

module configuration file: A configuration file that represents the connectivity of, and is composed of, one or more lower level modules and/or devices (i.e., a board, a collection of boards, etc.).

P

Parallel Signature Analysis: *See PSA.*

PRPG: *Pseudo-Random Pattern Generation.* An LFSR (Linear Feedback Shift Register) used to generate test data to the design.

PSA: *Parallel Signature Analysis.* An LFSR (Linear Feedback Shift Register) used to compress test data through collecting data from the design by exclusive ORing the resulting test data with register state data.

Pseudo-Random Pattern Generation: *See PRPG.*

R

RUNBIST: An 1149.1 optional instruction that causes execution of a built-in self-test operation on a component. (See BIST.)

Run-Test/Idle: An 1149.1 TAP controller state in which a Built-In Self-Test operation can be performed.

S

Scan Control Module: The Scan Control Module (SCM) is an IEEE 1149.1 hardware scan controller. The SCM is a half-slot, PC-AT-backplane type card that resides within the I/O address space. The board consists of the TBC (Test Bus Controller) IC and additional logic which is used by ASSET to perform all 1149.1 operations.

SCM: *See* Scan Control Module.

SCOPE: *System Controllability, Observability, and Partitioning Environment.* A structured hardware and software methodology that allows test engineers to build 1149.1 capability into designs for board-level or IC-level testing. SCOPE products are ideal for creating partitions and putting in-circuit test structures around complex devices such as microprocessors.

SCOPE octals: 1149.1-compatible devices that can replace standard logic functions, such as buffers and transceivers. These TI-designed devices have boundary scan cells at the perimeter of functional pins, except power and ground. When operating in a functional mode, 1149.1 circuitry of the octal devices is inactive, allowing those devices to perform exactly like their non-1149.1 counterparts. When the devices are placed into 1149.1 test mode, all inputs and outputs can be controlled and observed. ASSET includes a set of octal configuration files that allows use of ASSET in conjunction with the octal devices.

T

TAP: *Test Access Port.* The 16-state finite state machine which generates 1149.1 test logic control signals.

target system: Any UUT (Unit Under Test) which contains 1149.1-compatible devices.

target module: *See* UUT.

TBC: *Test Bus Controller.* A device or circuit that controls an 1149.1 test bus. The TBC issues commands and scans data in and out of 1149.1 devices. Only one TBC can be active on a 1149.1 bus at any time.

TCK: *Test Clock.* The clock input pin contained in the Test Access Port (TAP) used to clock test (non-functional) memory logic.

TDI: *Test Data Input.* The data input pin used to load data into the 1149.1 logic.

TDL: *Test Description Language.* A test pattern format for I/O connector pins which describes the stimuli and expected results for a test application. TDL can be used to interface CAD tools with ASSET.

TDO: *Test Data Output.* The The data output pin used to unload data from the 1149.1 logic.

Test Access Port. *See* TAP.

Test Bus Controller. *See* TBC.

Test Clock. *See* TCK.

Test Data Input *See* TDI.

Test Data Output. *See* TDO.

Test Description Language *See* TDL.

test logic: The electronic circuitry and computational operations of an octal device when it is in test mode, as opposed to normal (functional) mode.

Test Mode Select. *See* TMS.

test vectors: A set of input stimuli values that are applied at the beginning of a period of time and a set of output observation values made at the end of a period of time that are represented by a string of values. The order position corresponds to the specified pin order.

TMS: *Test Mode Select.* The mode select pin used to control the execution of 1149.1 instructions.

TRSTZ: *Test Reset, low.* The optional reset pin for the TAP (Test Access Port) controller.

U

Unit Under Test: *See* Unit Under Test.

UUT: *Unit Under Test.* A logical function or device that will be exercised by a tester.

Index

P

S

T